Paul Roper

Commodore Sachbuchreihe Band 14

# PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64:

## SPIELE

**Paul Roper**

# PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64: SPIELE

Commodore Sachbuchreihe Band 14

Paul Roper

# PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64: SPIELE

Commodore

Titel der Originalausgabe: Roper, Paul – Machine Code Games Routines for the Commodore 64.
Copyright © Paul Roper, 1984.
First published 1984 by
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street, London WC2R 3LD

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronical, mechanical, photocopying, recording, or otherwise without the prior permission of the publishers.

Aus dem Englischen übertragen von Peter Schönau und Jörgen Danielsen.

Copyright © der deutschen Ausgabe bei Commodore Büromaschinen GmbH, Frankfurt 1984.

Alle deutschsprachigen Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung von COMMODORE reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Nachdruck, auch auszugsweise, nur mit schriftlicher Genehmigung von COMMODORE.
Artikel-Nr. 556427/4.85 Änderungen vorbehalten ISBN-Nr. 3-89133-014-6

# INHALT

**Kapitel 1** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
**Einführung**
Der Vorgang des Spieleschreibens, Gliederung dieses Buches, Interrupts und der Commodore 64, Maschinencode und BASIC.

**Kapitel 2**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
**Weitere 6510-Theorie**
Arbeitsweise der Chips 6510 und 6502, Adressiertechniken, der Stapelspeicher, Mehr-Bit-Operationen, logische Operationen, Bildschirmabtastung, Multiplikation und Verschiebungen, Datenstrukturen und Tabellen.

**Kapitel 3**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
**Die Geburt eines Spieles**
Beginn des Spieleschreibens in BASIC und/oder Maschinencode, die wesentlichen Zutaten eines guten Spiels, Entwicklung Ihrer eigenen Spielidee.

**Kapitel 4**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
**Programmentwurf**
Aufstellung von Diagrammen zur Unterteilung eines Spiels in einer Reihe von kleineren Aufgaben, Betrachtung von Vereinfachungsdiagrammen, wie Diagramme beim Schreiben von Programmen helfen.

**Kapitel 5**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
**Programmcodierung**
Funktionsprogramme, Parameterübergabe, Programmkompilierung, Parameterkollisionen, Assembler.

**Kapitel 6**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
**Schreiben Ihrer eigenen Routinen**
Schreiben Ihrer eigenen Routinen, Umwandlung von Algorithmen in Maschinen-Coderoutinen, Entwicklung eines Programms zur spiralförmigen Bildschirmauffüllung, Algorithmusentwurf, der Ihr Gehirn stimuliert, Probleme zu lösen und neue Programme zu erzeugen.

**Kapitel 7** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
**Prüfung und Fehlerbeseitigung**
Fehlerursachen, Einfachhalten von Programmen, Programmprüfung, Fehlerbeseitigung.

**Kapitel 8** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
**Zeichenfolgen (Strings): Zusammensetzung des Spiels**
Welche Verbindungen herzustellen sind, Iteration, DO-UNTIL-Schleifen, bedingte Sprünge, Überprüfung von Fehlern in Bindegliedern.

**Kapitel 9** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
**Direkte Programme**
Text und langsamer Druck, Speicherauffüllung, Speicherkopie, Verzögerungen, Bombennachführung, Rollen, Interaktion, Umkehrung und Explosionen, alternatives Spritesystem.

**Kapitel 10** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
**Indirekte Programme**
Unterbrechung, Sprites, weitere Verwendungen von benutzerdefinierbaren Graphiken (UDG), Geräusche, Spiele mit großem Umfang, Flottenbewegungen, Zufallszahlen, nichtlineare Bewegung, Farbspeicher.

**Anhang A** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
**Dienstprogramme**
Wiederherstellung eines abgestürzten Programms, Programmplazierung und Speicherverwaltung, Eintreten in den Maschinencode von BASIC aus, benutzerdefinierbare Graphiken und ein UDG-Editor, Sprites und ein Spriteeditor.

**Anhang B** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
**Miniassembler**
BASIC-Assembler einschl. eines Disassemblers, Programmbeschreibung, Verwendung des Programms, Miniassembler-Programmausdruck, benutzerdefinierbare Funktionstasten, dynamischer Halt, Erleichterung für den Anführungsbetrieb, Geräusch- und Spritestummschaltung, kleiner Werkzeugsatz.

**Anhang C** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
**Höhere Maschinenkennwerte**
Die Tastatur und ihr Puffer, Text- und Variablenspeicherung in BASIC für CIAs, SIO und VIC Speicherbelegungspläne.

**Anhang D** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
**Laserbike**
Vollständiger Spielprogrammausdruck.

## VORBEMERKUNG DES AUTORS

Dieses ist eine Zusammenstellung von Programmen, Verfahren und Techniken, die alle bei der Konstruktion von Spielen nützlich sind. Ich hatte viel Spaß bei der Zusammenstellung dieser Liste – was ich vorher nicht ahnte. Es gibt viele Programme, die aus verschiedenen Gründen in diesem Buch nicht aufgelistet werden konnten. Die hier aufgelisteten sollten jedoch ausreichen, Ihnen einen gedanklichen Start in die Welt der Maschinencode-Spielprogrammierung zu geben. Haften Sie nicht zu stark an meinen Verfahren und Ideen: Entwickeln Sie sich nach Gutdünken. Ich bin ein vollständiger Autodidakt und habe daher meinen besonderen Stil.

Dieses Buch ist so ausgelegt, daß es mehr als nur eine Zusammenstellung von Programmen ist. Alle Ideen werden erklärt, so daß Sie sie getrost verändern, mischen und auseinanderreißen können. Sie sollen sie nicht nur eintippen und verwenden – lassen Sie sie Ihr wahres Können zeigen.

Aus Gründen der Loyalität beziehe ich mich im gesamten Buch sowohl auf den 6502 als auch den 6510. In Bezug auf Programmierung sind die beiden identisch. Ich habe den 6502 schon programmiert, lange bevor der C 64 auf dem Markt war, daher fühle ich mich zu ihm hingezogen.

Schließlich möchte ich meinen Freunden und meiner Familie danken, daß sie Nachsicht mit mir übten und hilfreiche Vorschläge machten, während ich das Buch schrieb.

## PROGRAMMANMERKUNGEN

In den Computerlistings in diesem Buch wurden einige Steuersymbole wegen der besseren Lesbarkeit ausgetauscht. Sorgen Sie bitte dafür, daß Sie die richtigen Zeichen eintasten.
Die in den Ausdrucken in eckigen Klammern erscheinenden Befehle werden unten aufgelistet:

```
BEZEICHNUNG  SYMBOL TASTEN   ODER   CHR$(XXX)
-----------------------------------------------
[SCHIRM NEU]     "■"  = <SHIFT>+<CLR>   CHR$(147)
[CRSR HOCH]      "■"  = <SHIFT>+<CRSR>  CHR$(145)
[CRSR RUNTER]    "■"  = <CRSR>          CHR$(17)
[CRSR LINKS]     "■"  = <SHIFT>+<CRSR>  CHR$(157)
[CRSR RECHTS]    "■"  = <CRSR>          CHR$(29)
[SCHWARZ]        "■"  = <CTRL>+<1>      CHR$(144)
[WEISS]          "■"  = <CTRL>+<2>      CHR$(5)
[ROT]            "■"  = <CTRL>+<3>      CHR$(28)
[CYAN]           "■"  = <CTRL>+<4>      CHR$(159)
[PURPUR]         "■"  = <CTRL>+<5>      CHR$(156)
[GRUEN]          "■"  = <CTRL>+<6>      CHR$(30)
[BLAU]           "■"  = <CTRL>+<7>      CHR$(31)
[GELB]           "■"  = <CTRL>+<8>      CHR$(158)
[REVERS EIN]     "■"  = <CTRL>+<9>      CHR$(18)
[REVERS AUS]     "■"  = <CTRL>+<0>      CHR$(146)
[ORANGE]         "■"  = <C=>+<1>        CHR$(129)
[BRAUN]          "■"  = <C=>+<2>        CHR$(149)
[HELLROT]        "■"  = <C=>+<3>        CHR$(150)
[GRAU 1]         "■"  = <C=>+<4>        CHR$(151)
[GRAU 2]         "■"  = <C=>+<5>        CHR$(152)
[HELLGRUEN]      "■"  = <C=>+<6>        CHR$(153)
[HELLBLAU]       "■"  = <C=>+<7>        CHR$(154)
[GRAU 3]         "■"  = <C=>+<8>        CHR$(155)
<CTRL> BEDEUTET 'CONTROL-TASTE'.
<C=>   BEDEUTET 'COMMODORE-TASTE'.
```

# KAPITEL 1

# EINFÜHRUNG

Die Sprache BASIC ist in zweifacher Weise bemerkenswert: Sie ist einfach zu lernen und anzuwenden, aber sie ist recht langsam. Der Maschinencode ist auch leicht zu lernen (obwohl er sehr viel schwerer zu schreiben ist), aber er ist erstaunlich schnell. Werden die beiden gemischt, dann entsteht eine Zwitterform der Programmierung: eine, die leicht zu verwenden, aber trotzdem schnell ist. Idealerweise werden die besten Programme in 100 % Maschinencode geschrieben, aber das Schreiben dieser Art Programme benötigt viel Übung und Erfahrung. Eine Mischung liefert die Vorteile beider Welten. Dieses Buch behandelt eine solche Verwendung des Maschinencodes in Spielen. Das Schreiben des reinen Maschinencodes ist einfach eine logische Erweiterung einiger der Ideen, aber seien Sie nicht ungeduldig, es kommt von allein.

Das Problem beim Maschinencode ist, daß man oft Vorurteile hat. Die Anweisungen selbst sind so lächerlich elementar, daß jede größere Aufgabe zumindestens beängstigend erscheint. Im Gegensatz dazu eignet sich BASIC gut zur Konstruktion von längeren, komplexeren Programmen.

Betrachten Sie es mal so. Sie können jetzt wahrscheinlich recht gut „in BASIC denken". Wenn Sie in der Schule anfangen, Französisch zu lernen, dann erscheint die Sprache sehr abstrakt, aber nach einiger Zeit fällt es Ihnen leichter, in der Sprache selbst zu denken. Das gleiche gilt für den Maschinencode. Nach einiger Zeit entwickeln Sie ein Gefühl für die Sprache. Das Mischen des Maschinencodes mit BASIC ist eine leistungsfähige Lernart. Je öfter Sie es tun, desto größer wird die Rolle des Maschinencodes in Ihren Spielen und obendrein benötigen Sie weniger Zeit, um ein anspruchsvolles Spiel zu schreiben. Wichtig sind die „Ausgeglichenheit des Spiels" und die Einfachheit der Maschinencode-Programmierung. Verwenden Sie am Anfang wenig Maschinencode und vergrößern Sie den Anteil nach und nach, wobei Sie den Maschinencode selbst einfach halten.

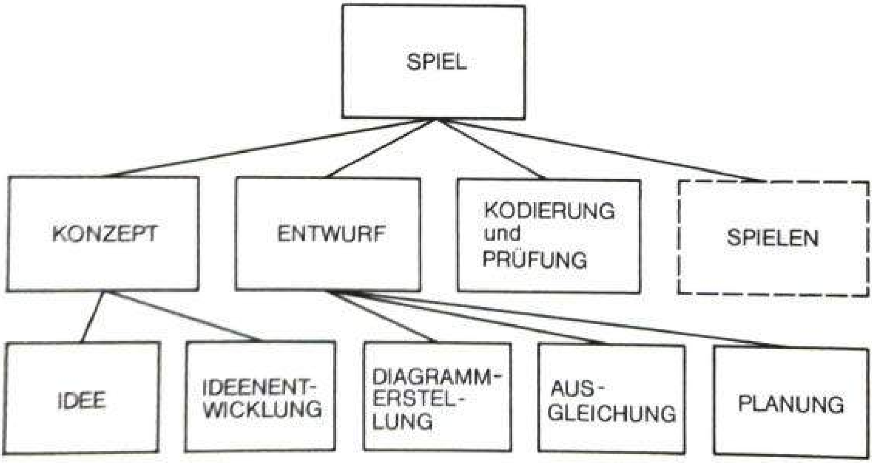
## DAS SCHREIBEN VON SPIELEN

Abbildung 1.1 illustriert den Schreibvorgang für ein Spiel. Von der ersten Idee an wird jede Stufe im Detail untersucht. Sie fangen links an und arbeiten stetig nach rechts. Wenn Sie da angelangt sind, haben Sie ein funktionierendes Spiel. Diese Diagrammart ist als Top-down-Diagramm (Vereinfachungsdiagramm) bekannt und wird in diesem Buch häufig benutzt.

Im Top-down-Diagramm beginnen Sie oben und arbeiten sich nach unten durch, wobei sich der Prozeß entwickelt. Das Ergebnis ist eine deutliche Darstellung der Prozeßstruktur.

Die erste Stufe ist das Konzept. Es mag Ihnen dumm erscheinen, eine Menge Zeit damit zu verbringen, über den Zweck des Spiels nachzudenken. Damit sich die beträchtliche Zeit und Mühe, die Sie für das Spiel aufbringen werden, lohnen, brauchen Sie eine gute Idee. Die Idee ist das grundlegende Prinzip des Spiels, egal ob es sich um das Apfelklauen oder die Zerstörung von Raumschiffen handelt. Danach nimmt die Idee durch eine Menge von Feinheiten Gestalt an. Diese können Mutterschiffe und Kraftpillen sein, die das Spiel beleben. Zwei Dinge dürfen bei einem Spiel nicht fehlen: gute Darstellung und der letzte Schliff; nehmen Sie sich also dafür Zeit.
Die nächste Stufe ist der Entwurf. Hier wird das Spiel in Diagrammen aufgezeichnet, damit wir sehen können, was gemacht werden muß. Danach kommt der Ausgleich, in dem das BASIC-Maschinencode-Gemisch festgelegt wird. Als nächstes beginnt die Planung, wobei die Maschinencode-Programme identifiziert werden. Beim Schreiben im Maschinencode müssen Sie sich auf jeden Fall Aufzeichnungen machen. Alle Diagramme sollten numeriert und die Programme sollten ordentlich zusammen mit ihren Funktionen aufgelistet werden. Die gesamte Dokumentation sollte sauber in einer Datei geführt werden. Die Dokumentierung ist wertvoll, wenn sie ordentlich gemacht wird. Es ist schön, wenn Sie am Ende des Tages eine ordentliche Datei besitzen, die alle Einzelheiten des Programms enthält. Im gesamten Buch beziehe ich mich auf die „Programmdateien", und Sie werden herausfinden, daß diese Dateien Ihnen sehr helfen, wenn Sie wissen wollen, wie Ihr Programm Daten aufnimmt und wieder abgibt.

**Abbildung 1.1: Der Spieleherstellungsprozeß**



Kehren wir zur Planung zurück. Dies bedeutet einfach, daß man auf die Programmdateien schaut und einen Ablauf für das Codieren, Prüfen und Dokumentieren plant, damit Sie Termine setzen können.
Das Codieren und Prüfen bringt das Spiel wirklich in Gang. Dies heißt, die Programme zu konstruieren und an ihnen herumzufeilen, bis sie funktionieren. Erst dann ist das Programm vollendet und das Spielen kann beginnen.

## DIE PROGRAMMROUTINEN

Wie der Titel sagt, enthält dieses Buch eine Fülle von Maschinenprogrammen. Diese führen eine Menge einfacher und komplexer Aufgaben aus. Sie sollen Ihnen dabei helfen, die Last von den BASIC-Programmen zu nehmen. Sie sind jedoch nicht dazu gedacht, daß Sie sie einfach kopieren und dann vergessen. Wollen Sie ein Programm verwenden, dann nehmen Sie die Mühe auf sich, und finden Sie heraus, wie es arbeitet. Es könnte sein, daß Sie Änderungen vornehmen wollen. Falls Sie das Programm nicht verstehen, werden Änderungen unmöglich. Da es unsere Absicht war, daß Sie die Programme verstehen, werden diese von einer Beschreibung des Themas und der Art, wie das Maschinenprogramm arbeitet sowie gelegentlich einigen Bemerkungen über den Programmierstil begleitet. Die Programme selbst sind mit kurzen Anmerkungen versehen. Der Programmablauf ist durch Pfeile gekennzeichnet, so daß Sie mit einem Blick Querverweise zu den zugehörenden Ablaufdiagrammen erkennen können. Diese Unterstützung bietet mehr als nur Programme, sie liefert auch die Verfahren und Techniken mit. Das Kapitel über die Programme ist in einen direkten und einen indirekten Abschnitt aufgeteilt. Der direkte Abschnitt enthält Programme, die scharf umrissene Aufgaben durchführen, wie z. B. das Auffüllen des Speichers oder das Blättern der Bildschirmanzeige, während der indirekte Abschnitt Konzepte behandelt (die von Programmen unterstützt werden), z. B. Bewegung einer Gruppe von Objekten über den Bildschirm.

## DAS BUCH

Nicht alle Programme dieses Buches sind in dem jeweiligen Abschnitt enthalten. Sie werden Sie überall versteckt finden, oft nur deshalb, um einen Punkt interessant zu machen. Einen vollständigen Programmindex finden Sie als Anhang, am Ende des Buches. Die Anhänge enthalten außerdem andere wichtige Informationen, wie z. B. die Verwendung der Funktionstasten sowie eine Sammlung von Hilfsprogrammen. Dahinter verbirgt sich die Idee, diesem Buch einen Anreiz für das Lesen mitzugeben und es nicht nur als trockenes Nachschlagewerk zu verwenden. Ich schlage vor, daß Sie zunächst das Buch komplett lesen, bevor Sie

mit einem Programm beginnen. Sie könnten sonst eine gute Idee für die Erstellung eines Programmes verpassen.
Lesen Sie das Buch nicht zu schnell durch. Ich habe eine Menge Zeit damit verbracht, Ideen nur so sprudeln zu lassen, der einzige Weg, um die Maschine und diese Programme richtig kennenzulernen. Spielen Sie mit ihnen, verändern Sie sie und zerpflücken Sie sie, bis Sie wirklich mit ihnen vertraut sind. Es wird Ihnen Spaß machen, eine Stunde fröhlich mit Ideen herumzuspielen. Wenn Sie nicht richtig mit dem Maschinencode vertraut sind, dann verstehen Sie die Kniffe nicht. Wenn sie Ihnen plötzlich gegenüberstehen, wissen Sie nicht, was passiert. Das kann sehr deprimierend für Sie sein – ein Computer ist ein sehr strenger Lehrer und erlaubt keine Fehler.
Wo wir gerade über Fehler reden, es gibt kein Maschinenprogramm ohne sie. Fehler führen normalerweise zu einem Programmabbruch. Keine Panik, wenn Ihnen dieses passiert – machen Sie sich nur Gedanken über Ihr Programm. Oft befreit Sie schon die Reihenfolge der Eingabe RUN/STOP zusammen mit RESTORE. Vergessen Sie nicht, ungefähr alle halbe Stunde eine Sicherungskopie zu machen und überschreiben Sie keine vorausgegangene Sicherungskopie. Sie würden die alte Sicherungskopie verderben und sich Schreibfehler in der neuen einhandeln. Überprüfen Sie alle Ihre Sicherungskopien. Das kostet bei einem Bandsystem Zeit, aber das ist es wert, sollte es einen Spannungsausfall geben. Ist die Maschine weiterhin blockiert, dann hilft Ihnen ein kleines Programm aus dem Anhang A (Absturzbeseitigung). Sehen Sie es sich einmal an.
Vergewissern Sie sich zuerst, ob die Programme, die Sie benötigen, nicht in dem Buch bereits vorhanden sind. Sie könnten versuchen, ein Spiel um ein oder zwei vorhandene Programme herumzuschreiben, anstatt die Programme um ein Spiel. Sie könnten beispielsweise das Programm, das Sprites in die Ausgangsstellung bringt, dazu verwenden und ein Jagdspiel schreiben, indem der Spieler ein Sprite mit einem Joy-Stick steuert und bestimmte Aufgaben auf dem Bildschirm ausführen muß, ohne mit einem in die Ausgangsstellung zurückkehrenden Sprite zu kollidieren (siehe hierzu Kapitel 10). Solch ein Programm ist leicht zu schreiben und läuft schnell, besonders dann, wenn es an den Interrupt (Programmunterbrechung) angehängt ist.

## INTERRUPTS

Interrupts sind ein wesentliches Werkzeug in der Ausrüstung des C 64. Vielleicht verstehen Sie den Begriff Interrupt jetzt noch nicht. Spätestens am Ende des Buches wird er Ihnen ein Begriff sein. Der Interrupt ist zweifellos der Schlüssel zur Maschine. In den beiden Programmkapiteln werden viele der Programme speziell dafür entworfen, mit dem Interrupt zu arbeiten, einschl. eines Melodiespielers.

Wenn etwas „unterbrochen" wurde, ruft der C 64 das Programm automatisch für Sie ab. So können Sie beispielsweise alle 8 Sprites auf dem Bildschirm herumsausen lassen, während Sie ein Programm mit einem blinkenden Cursor auflisten und aufbereiten. Dieses wird Ihre Freunde in Erstaunen versetzen. Alles was ich sagen kann ist nur, wenn Sie Interrupts nicht verwenden, werden Sie nicht von den Vorteilen Ihres C 64 profitieren. Ein paar Programme in Anhang B zeigen Ihnen die Anwendung der Funktionstasten für einige bemerkenswerte Dinge.

## MASCHINENCODE

Zum Abschluß dieses Einführungsabschnittes möchte ich ein paar wichtige Tatsachen herausstellen. Zunächst: Das Schreiben im Maschinencode benötigt mehr Zeit als im BASIC. Sie werden das Programmieren mit großem Elan beginnen. Danach folgt jedoch ein langer Zeitraum ohne greifbare Ergebnisse – die Talsohle. Viele „Möchtegern"-Programmierer geben in dieser Stufe auf, was zu unfertigen Programmen führt. Das Programmieren im Maschinencode erfordert Durchhaltevermögen, Zielstrebigkeit und Überzeugung. Fangen Sie mit einfachen Programmen an und arbeiten Sie sich zu den großen durch.

Ein weiterer, wichtiger Punkt ist, daß es praktisch eine unendliche Anzahl von Möglichkeiten gibt, ein Programm zu schreiben, das eine Aufgabe erfüllen soll. Meine Programme stellen einfach nur einen dieser Wege dar. Oftmals werden Sie sie verkürzen können. Denn sie sind so ausgelegt, daß sie Ihnen die Arbeitsweise zeigen und nicht mit „cleveren Tricks" prahlen, die reichlich im Maschinencode vorhanden sind. Die gleiche Denkweise gilt für die Programmierung von Spielen. Meine Anleitung ist nur als Richtschnur zu verstehen, anhand derer jeder seinen eigenen, besonderen Stil entwickeln kann. Warten Sie, bis Sie sicher sind, und entwickeln Sie dann Ihren eigenen, ganz persönlichen Programmstil.

Wenn Sie eigene Programme schreiben, halten Sie sie einfach. Wenn Sie sich die Pfeile in meinen Programmen ansehen, werden Sie feststellen, daß es nur wenige gibt und daß diese sich ineinander fügen. Werden wahllos Verzweigungen eingestreut, dann schleichen sich Fehler ein. Halten Sie Ihre Programme einfach. Oft kommen schlecht geschriebene Programme dadurch heraus, daß der Programmierer sie direkt über die Tastatur eingibt. Bei wirklich einfachen Programmen ist dies in Ordnung, aber ich habe die Erfahrung gemacht, daß der Bildschirm selbst nicht die Konzentration fördert. Viele meiner Programme werden zuerst auf dem Papier skizziert und dann übertragen. Da ich mir diesen Arbeitsstil anewöhnt habe, kann ich meine Programme überall schreiben (zu langes auf einen Fernsehschirm Starren ist sowieso schlecht für Sie).

An diesem Punkt muß die Frage nach Assemblern auftreten. Während die Assemblierung von Hand gut dazu geeignet ist, Grundlagen zu vermitteln, ist ein

Assembler für diese längeren Programme lebenswichtig. Am besten ist es, sich einen Steckkartenassembler zu besorgen. Ich persönlich habe mehrere Assembler, einschließlich eines „pensionierten" ARCON-ATOM, den ich manchmal parallel mit meinem 64 verwende. Wenn Sie von Hand assemblieren, dann hüten Sie sich vor Verzweigungsvektoren, sie sind nur schwer von Hand zu berechnen.

In Anhang B, am Ende des Buches, wird ein Assembler vorgestellt, der auch als Disassembler arbeitet. Egal, ob Sie einen Assembler haben oder nicht, der in diesem Buch dargestellte Adressiermodus (Tabelle 2.2 im nächsten Kapitel), ist wahrscheinlich nicht der, den Sie normalerweise verwenden. Der Grund dafür ist meine Bemühung, Sie dazu zu zwingen, die Programme zu verstehen, und sie nicht einfach blind einzutippen.

Ich lasse einige ziemlich harte Worte über BASIC in diesem Buch fallen, aber das soll Sie nicht von der Verwendung dieser Programmiersprache abhalten. BASIC bleibt die beste bisher erdachte Computersprache. Programme können schnell und wirksam zusammengestellt werden. Im gesamten Buch verwende ich BASIC, um einige Grundlagen zu demonstrieren. Lassen Sie BASIC nicht fallen, es ist großartig. Sind Sie bereit? Dann fangen Sie an, das Buch durchzulesen. Sie werden eine Menge nützliche Informationen finden, also lesen Sie weiter!

# KAPITEL 2

# WEITERE THEORETISCHE GRUNDLAGEN ZUM 6510-CHIP

Einige der in diesem Buch verwendeten Methoden mögen zuerst etwas seltsam erscheinen. Dieses Kapitel dient dazu, die Theorie, die hinter ihnen steckt, zu erklären. Wenn Sie die Arbeitsweise des 6510 und des 6502 bereits kennen, dann überspringen Sie dieses Kapitel. Es werden ziemlich viele Themen darin abgehandelt, und es ist für eine effektive Programmierung unbedingt nötig, daß Sie vollständig mit ihnen vertraut sind. Am besten spielen Sie diese Ideen durch, bis Sie sie instinktiv beherrschen. Sie meistern etwas erst, wenn Sie es tun, ohne überhaupt darüber nachzudenken.
Die hier vorgestellten Ideen sind meine eigenen. Benutzen Sie sie nicht einfach. Versuchen Sie mit 32 Bit- oder 64 Bit-Zahlen anstelle von 8 Bit- und 16 Bit-Zahlen zu arbeiten. Wenn Sie Ihre Fähigkeiten erweitern, ist Ihnen der Erfolg sicher.

## ADRESSIERVERFAHREN

Der Prozessor 6510 besitzt nicht weniger als 13 verschiedene Adressiermodi! Halten Sie einen Augenblick ein, und stellen Sie fest, an wie viele Sie sich erinnern können. Jeder vergessene Modus bedeutet eine verlorene Möglichkeit. Sie sollten nicht nur alle kennen, sondern auch in der Lage sein, sofort zu entscheiden, welches der beste für das anstehende Problem ist.

Tabelle 2.1 gibt alle Modi an. Wie viele kennen Sie wirklich?

Wahrscheinlich haben Sie diesen Test nicht allzu gut bestanden. Um wieder zu unserer früheren Analogie zurückzukehren, Adressiermodi sind wie Zeitformen im Französischen: Sie können mit einer oder mit zwei auskommen, aber nur so geradeeben. Lesen Sie weiter und entdecken Sie die Anwendungen der unbekannteren Modi.

**Absolut:** Der Absolut-Modus ist sehr wichtig. Er nimmt einen 16-Bit-Operanden und verwendet diesen als Adresse. Viele der anderen Modi bauen auf diesem auf. In diesem Buch wird der Absolut-Modus nur durch den Operanden bezeichnet, z. B.:

```
LDA 1024
```

welcher das erste Byte in dem Bildschirm liest (PEEKt). Ein naher Verwandter ist der ZERO-PAGE-Absolutmodus, der identisch ist, außer, daß er nur einen Ein-Byte-Operanden besitzt, der eine ZERO-PAGE-Adresse ist (mehr darüber später). Ein guter Assembler wird bei Bedarf automatisch diesen Modus verwenden, da er kürzer und schneller ist. Daher wird der ZERO-PAGE-Absolutcode genauso wie der normale Absolutcode dargestellt.

**Tabelle 2.1 Adressiermodi des 6510**

| | |
|---|---|
| ABSOLUT | Ein Zwei-Byte-Operand wird als 16-Bit-Adresse verwendet. |
| UNMITTELBAR | Ein Ein-Byte-Operand wird direkt verwendet. |
| IMPLIZIT | Einzel-Byte-Funktion mit eindeutiger Zuordnung. |
| ABSOLUT X | Wie ABSOLUT, jedoch wird das X-Register in die Adresse eingefügt. |
| ABSOLUT Y | Verwendet stattdessen das Y-Register. |
| ZERO-PAGE | Einzel-Byte-Adresse, die eine 8-Bit-Adresse im ZERO-PAGE angibt. |
| ZERO-PAGE X | Das X-Register wird hinzugefügt. |
| ZERO-PAGE Y | Das Y-Register wird hinzugefügt. |
| AKKUMULATOR | Einzel-Byte-Anweisung, die den Akkumulator betrifft. |
| RELATIV | Von den Verzweiungsanweisungen verwendeter Modus zur Erleichterung der Programmpositionierung. |
| VORINDIZIERT: | Einzel-Byte-Adresse im ZERO-PAGE, zu der das X-Register addiert wird. Die sich so ergebende Adresse wird mit dem PEEK-Befehl aufgerufen, „gePEEKt". Man erhält dadurch eine 16-Bit-Adresse. |
| NACHINDIZIERT | Einzel-Byte-Adresse im ZERO-PAGE. Sie wird „gePEEKt", um eine 16-Bit-Adresse zu erhalten. Danach wird das Y-Register addiert, um die Adresse zu indizieren. |
| INDIREKT | Ein 2-Byte-Operand wird „gePEEKt", um eine 16-Bit-Adresse zu erhalten. Er wird nur bei der JMP-Anweisung verwendet. |

UNMITTELBAR: Noch ein alter Bekannter. Der unmittelbare Modus besitzt immer ein Byte als Operanden, dessen Wert direkt verwendet wird. Es wird in diesem Buch durch das Symbol '#' gekennzeichnet. Dieser Modus besitzt keine weiteren „indizierten" Erweiterungen.

**IMPLIZIT:** Diese Einzel-Byte-Anweisungen führen einige Standardaufgaben durch. Sie werden nur durch eine entsprechende Drei-Buchstaben-Symbolik dargestellt.

**RELATIV:** Alle Sprunganweisungen, z. B.: BEQ, BCC und BNE verwenden die relative Adressierung. Unabhängig von ihrem Auftreten, geben sie der Maschine den Befehl, zu einer neuen Anweisung zu springen, indem gezeigt wird, wie weit und in welche Richtung diese von der Verzweigung entfernt liegt. Das Schöne daran ist, daß ein vollständig im Relativmodus geschriebenes Programm sich überall im Speicher befinden und laufen kann. Eine einzelne JMP-Anweisung innerhalb des Codes bedeutet, daß Sie die Sprungadresse neu berechnen müssen, wenn Sie verschieben. In den in diesem Buch enthaltenen Programmen habe ich überall die relative Adressierung verwendet. Die bereitgestellten Routinen sind 100%ig übertragbar. Sie können einen relativen Sprung erzeugen, indem Sie die entsprechende implizite Anweisung vor die Sprunganweisung setzen, selbst dann, wenn Sie nicht wollen, daß eine Bedingung wahr ist:

```
CLC
BCC
```
oder
```
SEC
BCS
```

Beide Anweisungen „erzwingen" die Verzweigung. Wenn Sie diese in einem Code sehen, werden Sie verstehen, was da vor sich geht. Einen Sprung an eine Stelle, die außerhalb des Sprungbereichs liegt, zeugt von einem schlechten Programmierstil. Können Sie dies nicht vermeiden, dann kehren Sie die Sprunganweisung um (d. h. BCC wird BCS und BPL wird BMI) und setzen Sie ein JMP dahinter. Sehen Sie sich dieses Beispiel an:

```
BEQ LOOP
RTS
```

Nehmen wir an, daß dieses zu einem Bereichsfehler (out of range) führt, so ist die Antwort wie folgt:

```
              BME STOP
              JMP LOOP
:STOP         RTS
```

Dies ist ein „erweiterter" Sprung und sollte einer „Bocksprung"-Technik vorgezogen werden, auch wenn die Auffindbarkeit verloren geht.

An dieser Stelle ist es angebracht, die Art, in der Variablen und Marken in diesem Buch dargestellt werden, anzusprechen. Vor Marken steht ein ':', und sie werden in einer Spalte deutlich links vom Code geschrieben. Variablen sind einfach kurze, aussagekräftige Buchstabengruppen, die einen Wert darstellen (z.B.: LDA CHAR besitzt CHAR als Variable). Sie können das auf zwei Arten und Weisen interpretieren:

- Entweder ist CHAR die Adresse des Wertes von CHAR und die Anweisung ist damit absolut
- oder CHAR ist der Wert selbst, die Anweisung wird dadurch unmittelbar.

Die Wahl bleibt vollständig Ihnen überlassen. Aus verschiedenen Gründen habe ich im gesamten Code LLX für Marken verwendet, wobei X irgendeine Nummer ist. Das Programm wird normalerweise mit :LL0 begonnen und :ILL1, :ILL2 usw. erscheinen in der Reihenfolge, in der ich sie bei der Kompilierung der Routine verwendet habe. Da jede Routine die gleichen Marken verwendet, sind Sie gezwungen, Ihre eigenen Marken zu entwickeln – eine Hilfestellung zum Erfolg.

**ZERO-PAGE – THEORIE:** In dieser Zusammenfassung der Adressierverfahren ist ein Wort zum ZERO-PAGE-Modus angebracht.
ZERO-PAGE ist der Kern des Computerspeichers. Es ist der Bereich, dem die Modi gewidmet sind. Deshalb sind diese Modi schnell und wirksam. Sie werden sehen, daß der C 64 dem Programmierer nicht viel Spielraum läßt, aber es ist ausreichend Platz vorhanden. Das Schöne am ZERO-PAGE-Modus ist, daß Sie nur einen Einzel-Byte-Operanden benötigen, um die 256 Bytes zu adressieren. Das Ergebnis ist mehr Platz, mehr Geschwindigkeit und größere Vielseitigkeit. Wenn Sie in das Handbuch schauen, dann werden Sie entdecken, daß der ZERO-PAGE-Modus voller interessanten Möglichkeiten steckt. Verwenden Sie also, wo immer möglich, Adressen im Bereich 2–255. Die Stellen 0 und 1 enthalten Anschlüsse und sollten deshalb freigelassen werden.

**ABSOLUT X und Y:** Dies sind enge Verwandte des Absolutmodus. Es wird eine 16-Bit-Adresse angegeben und zum Wert des X- oder Y-Registers addiert. Das bedeutet, daß eine Anweisung 256 unterschiedliche Adressen erzeugen kann. Dieser Modus wird u.a. dazu verwendet, sehr kurze Füll- und Rollroutinen zu erzeugen. Der Modus wird durch Anhängen von ',X' oder ',Y' an den Operanden dargestellt.
Die Anweisung LDA 1024,X könnte irgendeine Nummer von 1024 bis 1279 nur durch Änderung des X-Registers PEEKen. Es gibt auch ein Zero-Page-Gegenstück dafür, aber nur für das X-Register. (Ein Y-Registermodus ist vorhanden, er ist in seiner Anwendung aber sehr beschränkt.)

**INDIREKT:** Dieser Modus wird nur bei der JMP-Anweisung verwendet. In diesem Buch wird er nicht benutzt, aber er ist sehr handlich beim Aufnehmen eines Vektors. Es wird ein 16-Bit-Operand verwendet. Dieser zeigt auf ein Speicherstellenpaar. Dieses wird dann gePEEKt, und es erfolgt ein Sprung an die erhaltene Adresse. JMP (788) bedeutet: Nehme 788 und 789, verwende ihren Inhalt als 16-Bitadresse und springe zu dieser Adresse. Der Modus wird hierbei durch Einschließen der Adresse in ein Klammerpaar gekennzeichnet.

**INDIZIERTE ZERO-PAGE-MODI:** Diese Modi sind nützlich, da die wirksame Adresse so geändert werden kann, daß sie den gesamten Speicher des C64 umfaßt. Das Folgende passiert: Die Anweisung zeigt auf eine Page-Zero-Adresse. Der Wert, der in dieser Adresse gefunden wird, erzeugt eine 16-Bitzahl, die dann zur wirksamen Adresse wird. Obendrein wird durch Hinzufügen der Werte der X- und Y-Register mehr Flexibilität erreicht. Es gibt zwei Varianten. Bei der einen wird der Wert vorher und bei der anderen wird er nachher addiert. Die eine verwendet das X-Register und die andere das Y-Register.

**INDIREKT INDIZIERT:** Dies ist der Modus, der zum X-Register gehört. Der Operand ist eine Zero-Page-Adresse, zu der der Wert des X-Registers addiert wird (Indizierung). Die sich so ergebende Zero-Page-Adresse wird dann gePEEKt, und das Ergebnis des PEEK-Befehls liefert eine 16-Bit-Adresse (indirekt). Als praktisches Beispiel siehe Abb. 2.1.

**Abbildung 2.1 Indizierung und indirekte Adressierung**

Indizierung bedeutet, daß das X- oder Y-Register hinzugezählt wird.
Indirekte Adressierung bedeutet, eine Adresse wird gePEEKt, um auf eine neue Adresse hinzuweisen.
Der Unterschied zwischen indiziert indirekt (vorindiziert) und indirekt indiziert (nachindiziert) liegt in der Reihenfolge der Ausführung.
Das Laden des Akkumulators bei 100 im vorindizierten Modus bedeutet:

Akkumulator = PEEK(PEEK100+X)+PEEK(100+X)*256)

Das Laden des Akkumulators nachindiziert bei 100 bedeutet:

Akkumulator = PEEK(PEEK100)+PEEK(101)*256+Y)

Jeder der beiden Modi ist mit einem besonderen Indexregister verbunden.

**NACHINDIZIERT:** Dieses ist der Y-Register-Modus. Der indirekte PEEK-Befehl wird zuerst ausgeführt und dann wird die sich ergebende Adresse durch hinzufügen des Y-Registers indiziert. Diese beiden Modi sind bei der Handhabung von Datentabellen sehr nützlich, wie Sie in einem späteren Abschnitt sehen werden. Damit wird unser Überblick über mögliche Adressierverfahren abgeschlossen.

**Tabelle 2.2: Darstellung der Assemblersprache**

In diesem Buch werden keine Hexadezimalzahlen verwendet. Alle Zahlen sind dezimal. Diese Tabelle beschreibt die Darstellung aller in diesem Buch verwendeten Modi.

| MODUS | FORMAT |
|---|---|
| Impliziert | nur als 3-stelliger Buchstabencode |
| Akkumulator | „A" wird angehängt, z.B. ASLA |
| Absolut | Buchstabencode + Operand (Zero-Page-Adressierung über den Standard-Vorgabewert). |
| Absolut Y oder X | wie absolut, aber mit angehängtem „,X" oder „,Y". |
| Relativ | Buchstabencode + Marke |
| Nachindiziert | Operand in Klammern eingeschlossen mit angehängtem „Y". |
| Vorindiziert | Operand + „,X", in Klammern eingeschlossen |
| Unmittelbar | Operand mit vorangestelltem „#" |

## DER STAPELSPEICHER (STACK)

Der Stapelspeicher ist eine raffinierte Einrichtung, mit dem Sie JSR innerhalb JSR aufrufen und sich möglicherweise Ihren Weg mit RTS-Befehlen bahnen können. Das ist aber nicht seine einzige Verwendung.
Er kann auch angewendet werden, um Parameter weiterzugeben und Variable zu speichern. Zunächst wollen wir uns einmal ansehen, wie er arbeitet.
Der Stapelspeicher belegt Page 1 des C64-Speichers. Eine Hinweisadresse (Pointer) hält die Position innerhalb des Stapelzeichens aufrecht. Jedes Mal, wenn eine JSR-Anweisung angetroffen wird, dann wird die Rücksprungadresse in dem Stapelspeicher gespeichert, zur Herhinweisadresse 2 hinzuaddiert, und ein JMP (effektiv) zur Subroutinenadresse aufgerufen. Wird RTS angetroffen, dann wird die Hinweisadresse verringert, eine Adresse herausgezogen und ein JMP-Befehl zu dieser Adresse ausgeführt. Der Stapelspeicher umfaßt 256 Byte. Deshalb können 128 aufeinander folgende JSR-Befehle verschachtelt werden, bevor die erste Rücksprungadresse überschrieben wird: Versuchen Sie es lieber nicht, denn der Stapelspeicher wird noch für alle möglichen Dinge verwendet. Sie werden schon sehen, was passiert, falls er einmal voll werden sollte.

Die Anweisungen PHA und PLA bedeuten das Einschieben (PusH) und Herausziehen (PulL) des Akkumulators in bzw. aus dem Stapelspeicher. Wenn Sie etwas hineingeschoben haben, müssen Sie natürlich warten, bis die Hinweisadresse in der richtigen Position ist, bevor Sie es herausziehen können. Ein Verändern der Hinweisadresse selbst, kann mit Hilfe von TSX (Verschiebung der Hinweisadresse auf X) und CXS, die sie zurücksetzt, erfolgen. Seien Sie jedoch dabei vorsichtig, damit Sie keine Rücksprungadressen verändern.
Unter Anwendung dieser Ideen können wir den Wert im X-Register speichern und zwar innerhalb einer Routine ohne JSR, RTS oder anderer Speicheroperationen, durch Anwendung von TXA, PHA. Das Zurückladen erfolgt umgekehrt – PLA, TRX. Das ist zugegebenerweise nicht besonders elegant, aber wenn wir einen Parameter weitergeben wollen, kann es sehr hilfreich sein. Ist keine Page-Zero-Adresse vorhanden, dann bietet der Stapelspeicher eine bequeme Möglichkeit zur Zwischenspeicherung. Das Statusregister des Prozessors kann ebenfalls mit PHB und PLP im Stapelspeicher gespeichert oder zurückgeholt werden.
Tritt ein INTERRUPT auf, dann wird der Stapelspeicher dazu verwendet, alle Registerwerte und das Statusregister zu speichern, und beim Weiterlauf scheint es, als sei nichts geschehen.

## MEHR-BIT-OPERATIONEN

Die Addition von zwei 8-Bit-Zahlen ist einfach. Aber sobald wir mehr als 98 Bit verwenden, fangen die Dinge an, ein klein wenig komplizierter zu werden. Sie werden sicher wissen, wie man 16-Bit-Zahlen addiert. Dieser Abschnitt soll Ihre Erinnerung auffrischen und Ihnen zeigen, warum man es tut. Zuerst wollen wir uns die Addition ansehen. Haben Sie sie verstanden, dann schreiben Sie eine 64 Bit-Addierroutine, um Ihre Kenntnisse zu prüfen.
Als erstes muß die Übertragungsmarke gelöscht werden. Denn ADC bedeutet „ADd with Carry" (Addition mit Übertrag). Es wird eine Zahl in den Akkumulator addiert und, wenn die Übertragungsmarke gesetzt ist, wird zusätzlich eine 1 addiert. (Eine raffinierte Art der Addition zweier Zahlen unter Erhöhung des Ergebnisses ist die Verwendung eines SEC). Führt diese letzte Operation zu einem Überlauf im Akkumulator, dann wird die Übertragungsmarke auf 1 gesetzt. Werden die nächsten (höherwertigen) Bytes addiert, dann wird der Übertrag nicht gelöscht. Trat ein Überlauf auf, dann muß zusätzlich eine 1 addiert werden. Dieser Vorgang entspricht der Addition von Dezimalzahlen: Überschreitet eine Spalte die 9, dann müssen Sie eine 10 in die nächste Spalte übertragen, wo Sie als 1 erscheint. Für die gesamte Operation wird nur das erste CLC benötigt. Der folgende Code addiert zwei 16-Bitzahlen, die in 251 – 252 und 253 – 254 gefunden werden.

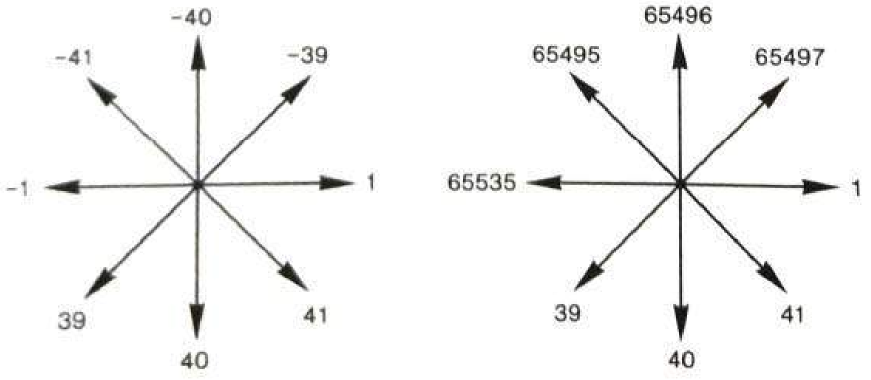
```
LDA 251
CLC
ADC 253
STA 253
LDA 252
ADC 254
SCA 254
```

Die Subtraktion wird genauso durchgeführt, nur wird vorher SEC verwendet, um den Übertrag zu setzen (SEt). Addition und Subtraktion sind bei Spielen von entscheidender Bedeutung, da sie es uns ermöglichen, Objekte über den Bildschirm zu bewegen.

Ein Vektor ist eine Zahl. Im Handbuch finden Sie Vektoren, die auf den Anfang von ROM-Routinen zeigen. In Verbindung mit Spielen ist ein Vektor eine Zahl, die zu einer Adresse hinzuaddiert wird, um eine Bewegung zu simulieren. Wenn ich von der Addition eines Vektors rede, meine ich die Addition einer Zahl zur Aktualisierung einer Position. Um z.B. einen Sprite nach rechts zu bewegen, addieren Sie den Vektor 1 zu seiner X-Koordinate. Per Definition besitzt ein Vektor sowohl eine Länge als auch eine Richtung. Beide können in einer Zahl enthalten sein: Ein Vektor für eine schnellere Bewegung nach rechts würde bedeuten, eine 3 anstelle einer 1 hinzuzuaddieren.

Abbildung 2.2 zeigt die 8 Richtungen und die zugehörenden Vektoren, die Sie in BASIC benutzen würden. Im Maschinencode wären getrennte Routinen für die Addition und Subtraktion reine Zeitvergeudung. Man erreicht dies, indem man sehr große Zahlen zur Subtraktion verwendet. Wenn Sie 65535 addieren, dann haben Sie bei 16 Bit effektiv 1 abgezogen.

**Abbildung 2.2: Subtraktion durch Addition**



Der zweite Teil der Abbildung sollte jetzt verständlicher sein. Anstelle von negativen Zahlen addieren wir große Zahlen. Sie sind in der Tat das Zweierkomplement von negativen Zahlen. Die Idee der Verwendung von großen Zahlen zur Durchführung der Subtraktion wird sehr oft verwendet. Sie sollten deshalb sichergehen, daß Sie diese Rechenmethode voll verstanden haben.

Der Vergleich zweier Zahlen auf Gleichheit oder Unterschiedlichkeit ist einfach. Sie verwenden dazu CMP oder (CPX oder CPX) und BEQ (gleich) oder BNE (unterschiedlich). Beim Arbeiten mit 16 Bit muß ein klein wenig mehr getan werden. Es kommt häufig vor, daß Sie prüfen müssen, ob eine Adresse erreicht worden ist (oder eine Rakete ein Schiff getroffen hat). Dafür wird ein 16-Bit-Vergleich benötigt. Der Vorgang ist einfach: Vergleichen Sie die beiden niedrigen Bytes und die beiden höheren Bytes getrennt. Stimmen die niedrigeren Bytes nicht überein, dann müssen die Zahlen unterschiedlich sein. Stimmen sie überein, dann überprüfen Sie die beiden höheren Bytes. Entweder stimmen sie überein (beide Zahlen sind gleich) oder nicht. Die folgende Routine tut dies:

```
LDA 251
CMP 253
BNE UNEQUAL
LDA 252
CMP 254
BNE UNEQUAL
**Numbers are equal**
```

Die beiden Zahlen werden in 251 – 252 und 253 – 254 gespeichert. UNEQUAL ist einfach eine Marke für Demonstrationszwecke.

Diese 16-Bit-Operationen sind sehr wichtig. Sie bilden das Herzstück beim Schreiben eines Spiels im Maschinencode. Ohne sie wäre es sehr schwierig, irgendwelche Objekte zu bewegen.

## LOGISCHE OPERATIONEN

Die Funktionen AND/OR/EOR erscheinen immer etwas fremd, da man keine direkte Anwendung erkennen kann. Sie machen aber die Behandlung einzelner Bits innerhalb eines Byte sehr einfach. Oftmals möchten Sie nur ein Bit in einem Byte setzen oder einige Bits an anderer Stelle ohne Störung des Restes löschen. Die Lösung dafür ist die Verwendung von logischen Operatoren. Jeder erfüllt seine eigene Aufgabe. Sehen Sie sich Abb. 2.3 an, um Ihre Erinnerung über ihre Funktion aufzufrischen.

**Abbildung 2.3: Wahrheitstabellen für AND, OR, EOR**

AND

| IN | IN | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Maske

OR

| IN | IN | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Einstellung

EOR

| IN | IN | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Umschaltung

**AND:** Das Ergebnis von AND ist dann (und nur dann) 1, wenn sowohl die erste Eingabe als auch die zweite Eingabe 1 ist. Daher ist 1101 AND 1100 gleich 1100. Die Funktion AND ist eine Maskierungsoperation. Sie ermöglicht Ihnen, unerwünschte Bits zu unterdrücken, während sie die wichtigsten Teile aufrechterhält. Nehmen wir an, daß wir an den niedrigeren 3 Bits einer 8-Bit-Zahl interessiert sind. Um nur diese Bits zu erhalten, maskieren wir mit 1+2+4=7. Nur wenn eine 1 in der Maske vorhanden ist, kann eine 1 herauskommen. Mit diesem Verfahren können wir eine ganze Menge anfangen: Überprüfen, ob eine Zahl gerade oder ungerade ist, oder ein Byte decodieren, das mehr als eine Information enthält. Die Funktion AND arbeitet als Sieb, durch das nur das Gewünschte fällt.

**OR:** Das Ergebnis von OR ist 1, wenn entweder die erste oder die zweite Eingabe 1 ist. Diese Funktion führt die Setzoperation (SET) durch. Wollen Sie ein Bit setzen, dann verknüpfen Sie es einfach mit der jeweiligen Zahl durch die Funktion OR. Verknüpft man eine Zahl durch die Funktion OR mit Null, dann bleibt sie unverändert, während sich bei 255 alle Bits der Zahl in Einsen verwandeln. Zugleich ist die Funktion AND eine Löschoperation (CLEAR). Wollen wir Bit 4 der Zahl 10101010 setzen, dann verknüpfen wir sie einfach über die Funktion OR mit 10000:

```
10101010
00010000
--------
10111010
```

Beim POKEn von wichtigen Stellen wie 0 und 1 ist es entscheidend, daß nur das von Ihnen gewünschte Bit geändert wird. Mißlingt dies, dann stürzt die Maschine mit Sicherheit ab.

**EOR:** Wie die Funktion OR, außer daß der Ausgang 0 ist, wenn beide Eingänge 1 sind. EOR ist eine Umschaltfunktion: Führt man nacheinander EOR-Verknüpfungen mit einer Zahl durch, dann gehen die entsprechenden Bits immer an und aus und wieder an. Das bedeutet, daß die zweimalige Durchführung von EOR die Zahl unverändert läßt. Man verwendet es zum Umschalten des Bit 7 eines Zeichens auf dem Bildschirm. Dieses schaltet dann ständig zwischen Positiv- und Negativdarstellung hin und her (blinken). Die EOR-Funktion kann auch zur Erzeugung des Zweierkomplements einer Zahl verwendet werden. Verknüpfen Sie dann die Zahl über EOR mit einer Zahl, die nur aus Einsen besteht und addieren Sie dann 1.

Mit diesen Operationen verfügen wir über ein Verfahren, jedes Bit zu setzen, zu löschen oder ein- und auszuschalten. Sehen Sie sich die folgende Anwendung für die Steuerung eines Bandmotors einmal näher an:

```
LDR 1
ORA #48
STA 1
```

schaltet ihn ab.
Dieses muß ständig wiederholt werden, um den Motor abgeschaltet zu halten.

## BILDSCHIRMABTASTUNG

Da dieses Buch speziell auf die Belange von Spielen zugeschnitten wurde, betreffen viele Routinen direkt den Bildschirm. Das bedeutet, daß diese den Bildschirm im Auge behalten und dessen Inhalt den unterschiedlichen Anweisungen entsprechend ändern. Deshalb sollte man schon einmal über den Vorgang der Bildschirmabtastung im Maschinencode nachdenken. Normalerweise wird der Bildschirm von oben nach unten abgetastet. Bei einigen Operationen ist jedoch eine umgekehrte Abtastung (die langsamer ist) nötig. Der Vorgang der Bildschirmabtastung verwendet die 16-Bit-Theorie, deshalb vergewissern Sie sich, daß Sie deren Wirkungsweise verstehen.
Die normale Bildschirmabtastung wird wie folgt durchgeführt. Die Adresse der oberen linken Ecke wird in einer Page-Zero-Adresse (zwei aufeinander folgende Bytes) gespeichert. Dann wird ein 16-Bit-Inkrement zur Erhöhung der Zahl und ein 16-Bit-Vergleichswert zur Bestimmung des Vorgangsendes verwendet. Die Funktion der Routine wird in die Schleife eingesetzt. Da sich die Adresse, unter der wir arbeiten, in der Zero-Page befindet, können wir den Bildschirm leicht abfragen. Die Anfangsadresse im LOW-HIGH-Format ist 1024:0,4.

```
LDA *0
STA 251
LDA *4
STA 252
```

Das Inkrement wird mit Hilfe der INC-Anweisung angegeben.

```
              INC 251
              BNE LLO
              INC 252
:LLO          -
```

Beim 16-Bit-Vergleich spielt es keine Rolle, ob wir zuerst das höherwertige oder zuerst das niederwertige Byte prüfen. Die Prüfung des niederwertigen Bytes ist aber schneller, da es nur bei wenigen Gelegenheiten „richtig" ist. Weiterhin müssen wir darauf achten, daß wir die Zahl prüfen, nachdem wir sie erhöht haben. Das bedeutet, daß wir auf das erste Quadrat **außerhalb** des Bildschirms prüfen müssen, da andernfalls die untere rechte Ecke außer acht gelassen würde.

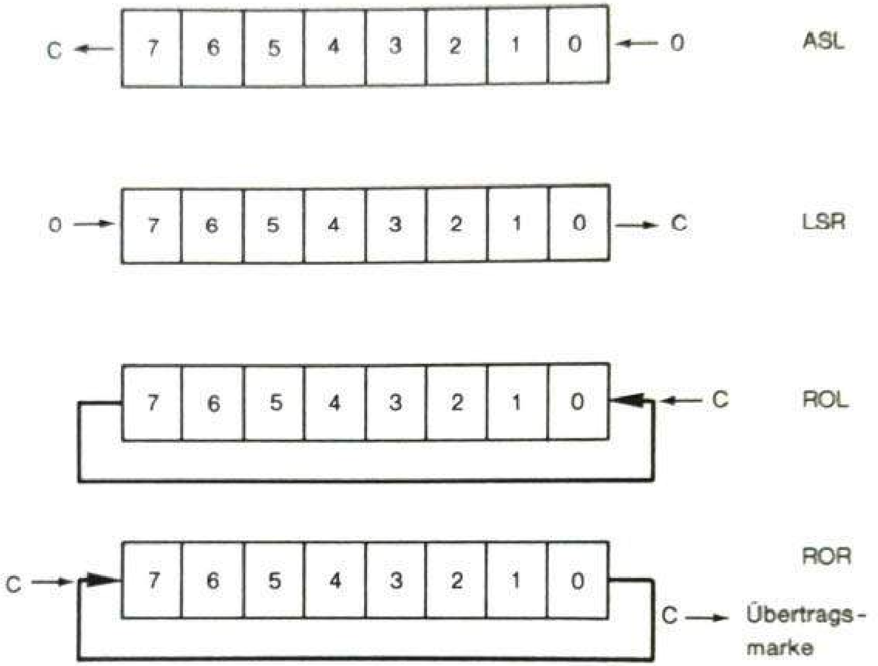
```
LDA 251
CMP *232
BNE [erneute Schleife]
LDA 252
CMP *7
BNE [erneute Schleife]
RTS
```

Wenn wir die Zahl erhöht hätten (d.h. sie für eine Aktivität auf dem Bildschirm verwendet und dann geprüft hätten), dann würden wir auch die letzte Adresse auf dem Bildschirm prüfen. Beim Programmieren in Maschinensprache müssen Sie auch auf solche Kleinigkeiten achten, die immer wieder auftreten.

## MULTIPLIKATIONEN UND VERSCHIEBUNGEN

Multiplikationen kommen im Maschinencode häufig vor. Ein typisches Problem könnte die Multiplikation eines Bytes mit 65 sein. Bei einem konstanten Multiplikanten ist das Problem leicht zu lösen. Variiert der Multiplikator, dann geht das Problem über den Umfang des Buches hinaus, obwohl das Prinzip das gleiche bleibt. Um z.B. mit 65 zu multiplizieren, teilen Sie die Zahl in Potenzen von 2 (64 und 1) auf. Die Multiplikation mit 2 ist einfach. Deshalb ist die Multiplikation mit einer Potenz von 2 genauso leicht.

**Abbildung 2.4: Verschiebungen und Rotationen**




Die Verschiebungsanweisungen ermöglichen uns die Multiplikation und Division der Zahlen in einem Byte. Abb. 2.4 zeigt, was passiert, wenn sie angewendet werden. Die ASL-Anweisung ist für die Multiplikation mit 2 besonders handlich. Wenn Sie es öfter tun, dann vergessen Sie nicht, einen eventuell auftretenden Übertrag festzuhalten. Abb. 2.5 zeigt, wie man das macht.
Das Ergebnis von A*B wird in T ausgegeben. Zuerst werden Kopien von A und B gemacht, um ihre Werte aufrechtzuerhalten. Dann wird die Multiplikation durchgeführt. Das erste Bit von X wird unter Verwendung der Funktion AND geprüft. Ist es eine 1, dann wird der Wert von Y zu dem Wert T addiert. Danach wird X durch 2 geteilt (nach rechts geschoben) und Y mit 2 multipliziert (nach links geschoben). Ist X = 0, dann wurden alle Bits aus X behandelt und die Aufgabe ist beendet. Ist X<>0, dann wird wiederum das erste Bit geprüft, aber es ist nicht das gleiche Bit wie vorher.
Würde man den gleichen Vorgang mit 16-Bit-Präzision ausführen, so wären dafür 16-Bit-Verschiebungen nach links und rechts, eine 16-Bit-Addition und ein 16-Bit-Vergleich erforderlich. In Wirklichkeit ist dies nicht so schwierig, wie es zunächst

erscheinen mag. Aber es ist unwahrscheinlich, daß Sie sich eine Multiplikationsroutine dieser Art wünschen. Was Abb. 2.5 also wirklich zeigt, ist die Wirkungsweise einer Multiplikation. Angenommen, wir wollen mit 40 multiplizieren. Ist A = 40, dann sind die einzigen in X gesetzten Bits 32 und 8. Deshalb addieren wir A x 2 dreimal (x 8) und B x 2 fünfmal (x 32). Die Multiplikation mit 2 erfolgt durch eine Verschiebeanweisung, es können ASL und ROL verwendet werden. Sie können ROL verwenden, um eine Mehrbitverschiebung wie folgt durchzuführen:

| | | |
|---|---|---|
| Löschen Sie zuerst den Übertrag | | CLC |
| Verschieben Sie als nächstes das erste Byte | | ROL ADDR |
| | od. | ASL ADDR |
| Verschieben Sie jetzt die nachfolgenden Bytes | | ROL ADDR + 1 |
| | | ROL ADDR + 2 |
| | | usw. |

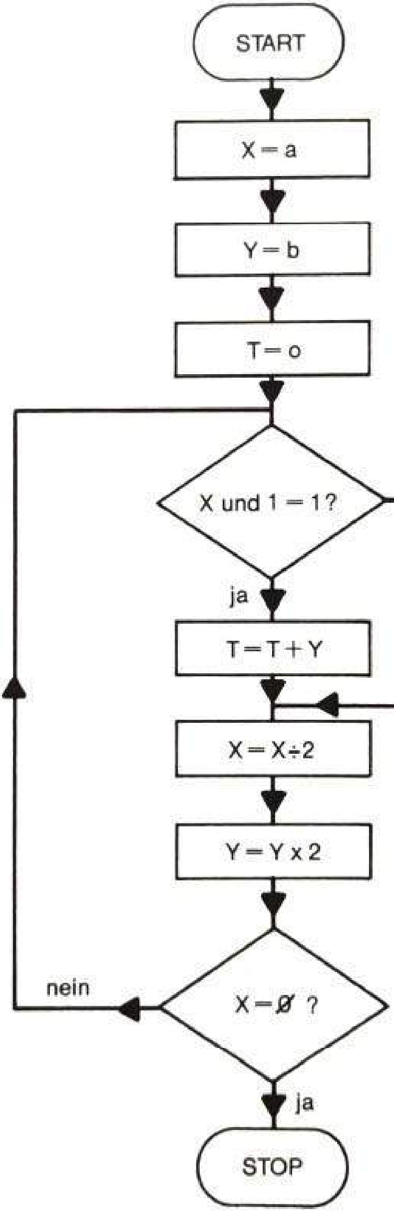
Verwenden Sie ROR zur Division durch 2 in einer Mehr-Bit-Verschiebung wie folgt:

| | | |
|---|---|---|
| Löschen Sie zuerst den Übertrag | | CLC |
| Als nächstes verschieben Sie das letzte Byte | | ROR ADDR + 2 |
| | od. | LSR ADDR + 2 |
| Verschieben Sie jetzt die nachfolgenden Bytes | | ROR ADDR + 1 |
| | | ROR ADDR |

## DATENSTRUKTUREN UND TABELLEN

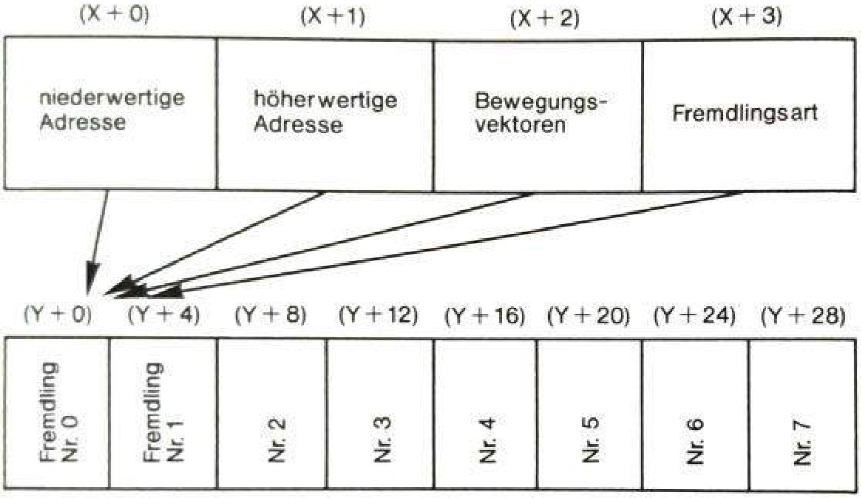
Die einfachste der BASIC-Datenstrukturen ist die Einzelvariable. Diese können leicht im Maschinencode gehandhabt werden. Aber es treten Probleme auf, sobald Tabellenfelder (Arrays) zur Kontrolle von Zeichengruppen benötigt werden. Die Verwendung dieser Art von Arrays wird in dem Abschnitt über Flottenbewegungen demonstriert (Kapitel 10). Ein Array ist nichts anderes als eine Liste von einfachen Variablen. Neulinge haben anfangs eine gewisse Scheu vor ihnen. Aber in Wirklichkeit sind sie bemerkenswert einfach zu handhaben.
Als erster Schritt ist das Array zu dimensionieren. Welche Informationen wollen Sie für jedes Objekt speichern? Einige Informationen, z.B. eine Adresse, benötigen 2 Bytes, wohingegen andere nur eines brauchen. Angenommen, wir benötigen eine Tabelle aus 50 Fremdlingen, die eine Adresse, einen Bewegungssektor und eine Fremdlingsart enthalten. Dies kann durch die in Abb. 2.6 gezeigte Struktur dargestellt werden. Jede Eingabe besteht aus 4 Bytes. Als nächstes entscheiden wir, wo die Tabelle untergebracht wird. Wir suchen einen geeigneten Speicherbereich.

**Abbildung 2.5: Durchführung der Funktionen T = A*B**



Speichern Sie den Tabellenbeginn als 2-Byte-Sektor an irgendeiner Stelle (auf Zero-Page, wenn möglich) und die Anzahl der Objekte (d.h. der Eingaben) irgendwo anders. Die Auffüllung der Tabelle mit geeigneten Daten ist eine Aufgabe für eine Copy-Routine. Das bedeutet, daß Sie die Angriffswelle 2 usw. leicht aufrufen können, indem Sie einfach die Anweisung geben, die Daten aus verschiedenen Speicherbereichen zu holen.

**Abbildung 2.6: Eine typische Tabellenstruktur**




Haben wir erst einmal die Struktur definiert, dann ist die Handhabung einfach. Nehmen wir an, die Basis der Tabelle liegt in 251 und 252 , dann können wir mit

```
LDY #0
LDA (251), Y
```

das erste Byte der Tabelle erhalten. Ist die Struktur wie oben angegeben, dann ist dieses das niederwertige Byte einer Objektadresse. Unter Verwendung von INY können nachfolgende Bytes aufgegriffen werden. Ist die Gesamtlänge der Tabelle kleiner als 256, dann wird die Abtastung über das Y-Register durchgeführt. Ist sie größer, dann ist es ratsam, Zahlen zu den Adressen 251 und 252 hinzuzuzählen und das Y-Register zum Lesen einer jeden Eingabe zu verwenden. Als Beispiel wollen wir annehmen, daß die Objekte in den Tabellen von 0 – 49 laufen und wir die Art des Fremdlings Nr. 12 herausfinden wollen. Da jedes Objekt aus 4 Bytes besteht, beginnt das Objekt, das uns interessiert, bei Tabelle +12 *4.

Nehmen wir an, daß 12 im Akkumulator ist. Zwei ASLA-Befehle multiplizieren mit 4 und TAY bewegt den Wert in Y. Die Fremdlingsart ist das vierte (und letzte) Byte des Objektes. Deshalb folgen 3 INYs. Ein LDA (251), Y liefert dann die Art des Objektes Nr. 12.

So erhält man den Objekttyp im Akkumulator

```
ASLA
ASLA
INY
INY
INY
LDA (251),Y
RTS
```

Wie Sie sehen, sind Tabellen sehr leicht zu benutzen. Sie werden sie in mehreren Routinen dieses Buches verwendet finden, einschl. der Bildschirm-Blink-Zuweisungsroutine (Invertierung, Umkehrung und Explosionen, Kapitel 9). Bei ihr wird eine Tabelle mit 25 Elementen verwendet, um festzulegen, ob eine Zeile blinken soll oder nicht.

# KAPITEL 3

# DIE GEBURT EINES SPIELES

Sie haben sich also entschieden, ein Spiel im Maschinencode zu schreiben. Sicher haben Sie bereits eine grobe Idee, wie das von Ihnen gewünschte Spiel ablaufen soll. Sei es nur eine weitere Variation des Space-Invader-Themas oder eine völlig neue Idee von Ihnen. Wie auch immer, hetzen Sie nicht durch diese erste Stufe. Bevor Sie überhaupt daran denken, zum nächsten Teil überzugehen, bringen Sie Ihre Gedanken in geordneter und übersichtlicher Form zu Papier. Auf keinen Fall können Sie ein Spiel entwerfen, wenn sich die Grundidee von Tag zu Tag nach Lust und Laune ändert. Kurz gesagt, entscheiden Sie sich, solange dies möglich ist. Haben Sie erstmal mit dem Programmieren begonnen, dann bezahlen Sie Änderungen teuer, sowohl im Hinblick auf die Zeit, als auch die Arbeitsmoral. Da wir gerade von Arbeitsmoral sprechen, sie ist eine der wichtigsten Faktoren beim Programmieren im Maschinencode. Es gibt lange Zeiträume intensiver Arbeit ohne sichtbares Ergebnis. Es ist ein harter Test des Durchhaltevermögens und des Selbstvertrauens. Aus diesem Grunde ist es für den Anfänger nicht ratsam, zu versuchen, ein Spiel vollständig im Maschinencode zu schreiben. Verwenden Sie den Maschinencode zunächst nur, um langsame BASIC-Abläufe in einem Spiel zu beschleunigen. Vielleicht finden Sie heraus, daß nur ein oder zwei Routinen nötig sind, um ein von Ihnen geschriebenes Spiel beträchtlich schneller zu machen. (In der Tat ist folgende Vorgehensweise, eine Spielidee im Maschinencode zu realisieren, sehr beliebt. Man schreibt zuerst ein BASIC-Programm, das die Aufgabe erfüllt, und wandelt dieses dann stufenweise in Maschinencode um. Ich mag dieses Verfahren nicht, wenn es dazu verwendet wird, ein reines Maschinencode-Programm zu erzeugen. Es führt zu ziemlich schlecht geschriebenen, schwer aufzubereitenden und wenig leistungsfähigen Programmen. Es kann jedoch erfolgreich in kleinem Umfang innerhalb eines Programmes Verwendung finden.)
Für den Anfänger ist es am besten, zunächst ein BASIC-Programm zu schreiben, wobei die Abschnitte ausgelassen werden, die im Maschinencode ausgeführt werden sollen. Der Grund dafür ist, daß BASIC und der Maschinencode zwei völlig verschiedene Sprachen sind. Der Versuch einer wörtlichen Übertragung ist zwecklos. Sie müssen zum ursprünglichen Problem zurückkehren, um rationell zu schreiben.
Ein klassisches Beispiel ist der Versuch, einen Computer dafür zu programmieren, daß er eine (menschliche) Sprache in eine andere übersetzt. Als Versuch wurde dem Computer die Aufgabe gestellt, einen Bibelabschnitt in eine andere Sprache zu übersetzen und dann diese wiederum in eine andere Sprache zu übersetzen. Schließlich wurde dieser Text dann noch einmal in eine andere Sprache übersetzt.

Der ursprüngliche Text war: „Der Geist ist willig, aber das Fleisch ist schwach". Als Endprodukt erhielt man den Text: „Der Alkohol ist gut, aber das Fleisch ist schlecht". Dieses sollte als abschreckendes Beispiel dienen, eine wörtliche Übertragung anstelle einer Betrachtung des Hauptproblems zu setzen.

## WIE MAN GUTE SPIELE SCHREIBT

Dieser Abschnitt ist dem Entwurf eines Spiels gewidmet, das herausfordern und süchtig machen soll. Es kann sein, daß man stundenlang über verschiedenen Ideen und Konzepten brütet, ohne eine zufriedenstellende Antwort zu erhalten. Eine sorgfältige Studie der beliebten Spielhallen-Spiele legt verschiedene Prinzipien frei, auf denen diese Spiele beruhen. Alle Spitzenspiele haben zu einem gewissen Ausmaß diese Qualitäten. Ich bin sicher, daß es möglich ist, von diesen Ideen zu profitieren, um ein erfolgreiches Spiel zu erzeugen.

Der erste Gesichtspunkt ist die Einfachheit. Die meisten Spielhallen-Spiele beruhen auf sehr elementaren Ideen. Sie garantieren einem Anfänger ein gewisses Maß an Erfolg und verhindern, daß er dauernd in den Regeln nachsehen muß. Die Idee kann alles sein, vom Sammeln von Äpfeln bis zum Bergsteigen. Neuere Ideen hat man mit dümmlichen Titeln wie „Angriff der mutierenden Hamburger" bedacht, aber erlaubt ist, was gefällt.

Es ist aber eine neue Generation von Spielen im Kommen, zu dem „Defender" gehört. Es ist ein sehr komplexes Spiel (was man schon an der Anzahl der Regler erkennen kann), aber es ist sehr befriedigend, wenn man es beherrscht. Es sollte jedoch jedem klar sein, daß ein komplexes Spiel schwieriger zu schreiben ist als ein einfaches. Versuchen Sie sich deshalb nicht an solch einem Kaliber, bevor Sie sich Ihrer Sache ganz sicher sind.

Der nächste Gesichtspunkt des Entwurfs ist die Abwechslung. Erneut müssen wir zwei Arten von Spielen betrachten – die alten und die neuen. Die alten Favoriten liefern endlose Variationen von Möglichkeiten, sie zu spielen: Bei „Space Invaders" z.B. können die Eindringlinge in jeder vom Spieler gewünschten Reihenfolge abgeschossen werden, wobei jede Reihenfolge zu einem anderen „Endspiel" führt. Die neue Generation von Spielen bietet Abwechslung über eine Reihe von verschiedenen Bildschirm-Bildern oder „Angriffswellen". Jedes Bild bietet eine neue, unbekannte Herausforderung und oft kann die Aufgabe auf mehrere Weisen gelöst werden. Durch Kombination der fast unendlichen Vielfalt mit der Einfachheit erhält man ein sehr gutes Spiel, das schnell zu begreifen ist und außerdem das Interesse aufrechterhält.

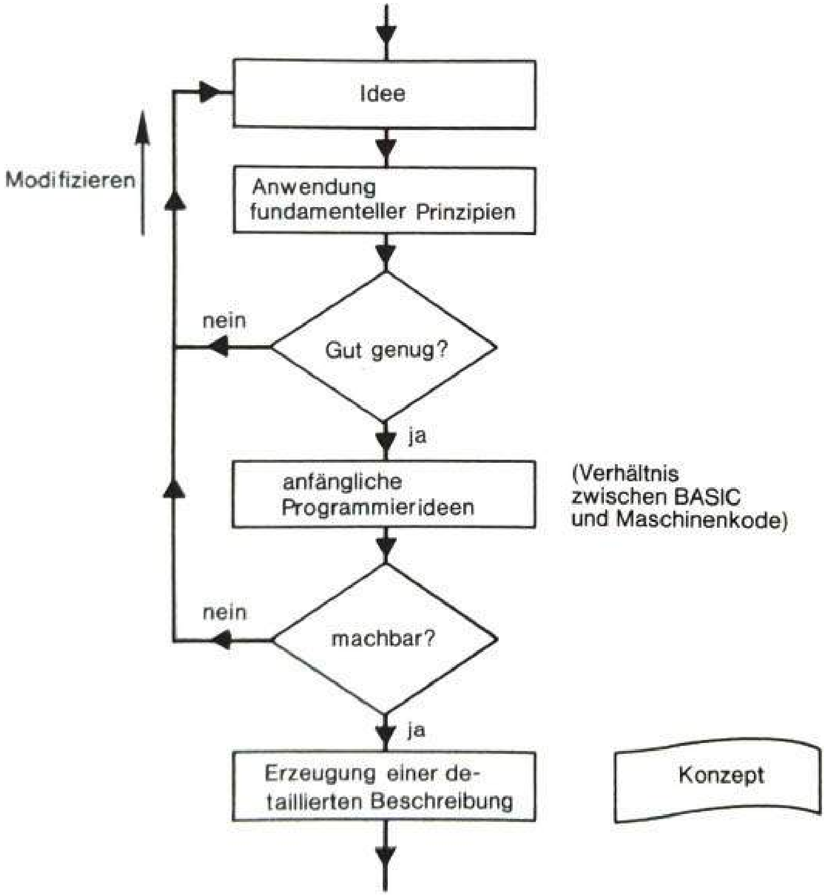
Ein dritter Gesichtspunkt ist die Einführung eines Elementes, das von Zeit zu Zeit dem Spiel eine Menge hinzufügt (wie z.B. ein Mutterschiff oder Früchte im Pac-Man-Labyrinth). Normalerweise liefert dieses Element eine schnelle Vergrößerung

des Spielstands auf Kosten zusätzlicher Gefahren. Die Ankunft eines Hetzers in „Defender" erhöht die Spannung und hält das Interesse an diesem gut ausgeführten Spiel wach. Dieses sind oft Stolpersteine für Spieler, die gerade ein Spiel in den Griff bekommen haben, bis ein neues Element hinzukommt und sie hinauswirft. In BASIC wird dieser dritte Gesichtspunkt wegen des starken Geschwindigkeitsverlustes zur Aktionssteuerung oft vernachlässigt.
Zweifellos ist ein allmählicher Anstieg des Schwierigkeitsgrades eine Haupteigenschaft in einem Spiel. Dies ist der vierte Gesichtspunkt. Beachten Sie jedoch, daß Schwierigkeit alleine nicht das Spiel ausmacht. Sie dient nur als Hindernis für den Spieler und stimuliert den Willen des Spielers, besser zu werden.
Ein Anstieg des Schwierigkeitsgrades kann auf zwei sinnvolle Arten und Weisen erreicht werden. Wird die Geschwindigkeit langsam gesteigert, dann muß der Spieler schneller denken und mehr koordinieren, was unausweichlich zu Fehlern führt. Der Geschwindigkeitsanstieg kann jedoch nur bis zu einem gewissen Grad gesteigert werden, da er sonst zu einem reinen Zufallsspiel führt. Wird die Intelligenz des Gegners während des Spielvorgangs vergrößert, dann wird der Spieler mit steigenden Problemen konfrontiert.
Dieses erreicht man manchmal durch eine Änderung der Feindbewegung von einer rein zufälligen in eine geordnetere Art. Durch Verbindung eines Geschwindigkeitsanstiegs mit einer Änderung der Reaktionen des Feindes ergibt sich eine natürliche Barriere im Spielablauf, die der Spieler nur durch mehr Übung überschreiten kann. Ein Versuch, dieses in BASIC zu erreichen, scheitert meistens kläglich: Eine Geschwindigkeitssteigerung kommt nicht in Frage, da das Spiel wahrscheinlich bereits mit voller Geschwindigkeit läuft und die Erhöhung der Intelligenz des Gegners ist nicht ratsam, da dies das Spiel noch weiter verlangsamen würde. Das führt dann zur Verwendung von albernen Verfahren (z. B. der Verdreifachung der Bombenzahl, die abgeworfen wird), um den Spieler zu besiegen. Der einzig vernünftige Weg ist die Verwendung des Maschinencodes.
Diese 4 Punkte (Einfachheit/Abwechslung/zusätzlicher Faktor/Schwierigkeit) sind die wesentlichen Zutaten eines guten Spiels. Um eine erfolgreiche Spielidee auszuknobeln, müssen Sie zuerst ein grundlegendes Ziel haben. Aber machen Sie es nicht zu kompliziert. Als nächstes denken Sie sich Erschwernisse für den Spieler aus. Gestalten Sie diese ebenfalls einfach. Führen Sie dann Abwechslung in das Spiel ein. Geben Sie dem Spieler eine Reihe unterschiedlicher Gegner und Aufgaben mit mehreren Lösungsmöglichkeiten. Sind Sie damit fertig, dann besitzen Sie die Grundlage des Spiels. Jetzt müssen Sie nur noch zusätzliche Elemente von Zeit zu Zeit einführen und einen Weg finden, die Schwierigkeit des Spiels langsam zu erhöhen. Ungeachtet dessen, was ich gesagt habe, erfordert der Entwurf eines Spiels Phantasie und Unvoreingenommenheit. Ideen kommen oft spontan. Seien Sie deshalb bereit, auf sie einzugehen. Wenn Sie einmal eine gute Idee haben, dann verschwenden Sie sie nicht. Geben Sie ihr die Zuwendung, die sie verdient und entwickeln Sie aus ihr ein ausgewachsenes Spielhallen-Spiel.

## ENTWICKLUNG DER IDEE

Haben Sie den genauen Spielablauf erst einmal festgelegt, dann sind Sie reif, zum nächsten Stadium des Programmentwurfs überzugehen. Aber zuerst sollten Sie ein paar Gedanken daran verschwenden, wie Sie Ihre Ziele erreichen wollen. Seien Sie realistisch. Obwohl der C64 außergewöhnliche Graphikfunktionen besitzt, bedeutet dies nicht, daß es leicht ist, ein Spiel mit viel Aktion auf dem Bildschirm zu schreiben. Tatsächlich ist es komplizierter als üblich. Zum einen wegen der Komplexität der Graphikerstellung und zum anderen wegen der Tendenz der Programmierer, mit zu ehrgeizigen Ideen zu beginnen. Alles was Sie in diesem Stadium wirklich brauchen, ist es, darüber nachzudenken, wie eine Sache in

**Abbildung 3.1: Konzept**



Beziehung zu einer anderen gebracht werden muß. Sie können sich z.B. dafür entscheiden, eine Sorte von Gegnern mit UDGs darzustellen, während die andere Sorte mit Hilfe der Sprite-Graphik-Funktion gezeigt wird.
Das sollte Ihnen dabei helfen, mögliche Bereiche für den Einsatz des Maschinencodes zu bestimmen. Sie werden die Spiele wahrscheinlich nicht 100%ig im Maschinencode schreiben, es sei denn, Sie haben bereits eine Menge Erfahrung. Sie können BASIC dazu verwenden, mit dem Spiel zu beginnen und dem Spieler Anweisungen zu geben. Möglicherweise wird nur ein kleiner Teil des Spiels vom Maschinencode unterstützt. Letztendlich bestimmt das Ausmaß des Springers zwischen den beiden Programmiersprachen die Zeit, die Sie brauchen werden, um das Spiel zu vollenden.
Ein übliches Verfahren zur Erzeugung überdurchschnittlicher Spiele ist, sie in zwei Versionen zu schreiben. Die erste bildet den Prototyp und wird so gut wie technisch möglich gemacht. Dann sollte sich der Programmierer eine Zeit lang bemühen, die ursprüngliche Spielidee weiter zu entwickeln. Dieses Nachdenken sollte sich in der zweiten und endgültigen Version wiederspiegeln. Dieses Verfahren hat jedoch den schwerwiegenden Nachteil, daß es eine Ewigkeit dauert, das Programm zu schreiben. Das kann man vermeiden, indem man das erste Programm vollständig in BASIC schreibt und dabei die fehlende Geschwindigkeit ignoriert. In diesem Stadium (das vielleicht nur ein oder zwei Tage dauert) werden sowohl die Idee als auch die beteiligten Verfahren entwickelt, so daß das Endprodukt so perfekt wie möglich wird. Dem Anfänger würde ich raten, daß er (oder sie) von diesem Verfahren absieht. Es ist für den Neuling langweilig und liefert ihm keine Erfolgserlebnisse.
Wenn Sie glauben, daß Sie Ihre Idee ausreichend im Detail entwickelt haben, dann vergewissern Sie sich, daß sie nicht zu ehrgeizig ist und (was noch wichtiger ist) daß sie zum späteren Nachschlagen in einer sauberen Form dokumentiert ist. Prüfen Sie nach, daß Sie das erreicht haben, was Sie ursprünglich wollten. Gleich werden Sie ziemlich viel Arbeit in dieses Programm investieren. Deshalb sollten Sie sicher sein, daß Sie auch das tun, was Sie tun wollen und nicht in irgend etwas anderes verstrickt werden. Maschinencode ist wirklich eine Sache des Durchhaltevermögens und der Tatkraft. Wenn Sie das, was Sie machen wollen, nicht wirklich interessiert, werden Sie das Spiel nie beenden oder auch nur dem Ende nahe kommen. Sie müssen mit dem gesamten Spiel vollständig vertraut sein, um seinen Fortbestand sicherzustellen. Wenn Sie sicher sind, daß Sie dieses Stadium zufriedenstellend abgeschlossen haben, dann gehen Sie zum nächsten Abschnitt über. Aber machen Sie zuerst eine Pause, damit die Idee vollständig in Ihrem Kopf reifen kann. Was Sie auch immer tun, überstürzen Sie es nicht.

# KAPITEL 4

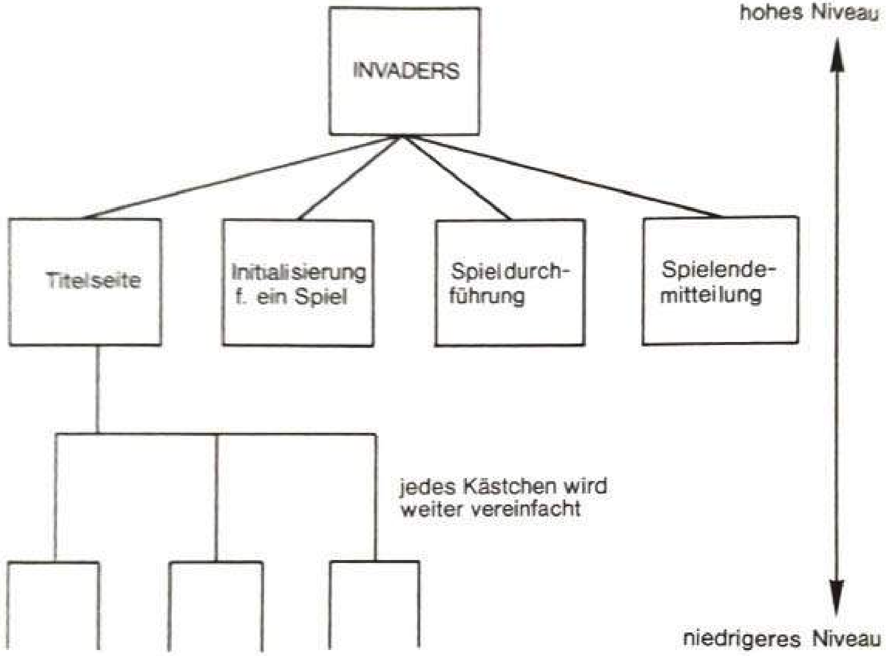
# PROGRAMMENTWURF

Ist die detaillierte Spielebeschreibung einmal fertiggestellt, dann werden in der nächsten Stufe Diagramme gezeichnet, die das Spiel in eine Reihe kleinerer Aufgaben aufteilen. Sie dürfen gerne fragen, wie diese Arbeit zu machen ist, aber ohne sie würde das Spiel niemals fertiggestellt.
In BASIC kann man Programme einfach aus dem Kopf direkt aufschreiben. Im Maschinencode ist die Situation anders. Bei einem längeren Programm kommt man um diese detaillierte Darstellung nicht herum. Auch wenn Sie nur einen Abschnitt des Programms im Maschinencode schreiben wollen, würde ich Ihnen dazu raten, sich an diese Stufen zu halten. Wenn Sie sich erst einmal damit befaßt haben, werden Sie überrascht sein, wie einfach es ist.
Es gibt im wesentlichen zwei Arten, einen Vorgang graphisch darzustellen: das Datenflußdiagramm bzw. der Programmablaufplan oder ein von oben nach unten gegliedertes Baumdiagramm. Jedes hat Vorteile, obwohl die Anwendung von Programmablaufplänen von einigen Institutionen als schlechte Programmierung angesehen wird. Mein System ist, ein sauberes Blatt Papier zu nehmen und oben in der Mitte der Seite den Titel des Programms in ein Kästchen zu schreiben. Dieses ist die „allerhöchste Ebene". Was ich damit sagen will ist, daß ich (der Mensch) es versteht, aber der Computer nicht die leiseste Ahnung von der Aufgabe hat. Der nächste Schritt ist, dieses Kästchen in eine Anzahl von nicht mehr als 4 oder 5 Aufgaben zu unterteilen. Als Beispiel wollen wir das Programm „Space Invaders" betrachten.
Ich würde dieses in folgende Teilaufgaben unterteilen:

- Anzeige der Titelseite.
- Spielbeginn.
- Durchführung eines Spiels.
- Spieler informieren, daß das Spiel vorbei ist.

Diese würde dann in eine Reihe von Kästchen unterhalb des ersten Kästchens gesetzt und wie in Abb. 4.1 verbunden. Beachten Sie, daß die Ebene der Beschreibung abgesenkt wurde und „maschinenorientierter" ist, als das erste Kästchen. Weitere Stufen bedeuten, daß jedes dieser Kästchen wiederum immer weiter unterteilt wird, bis die Codieraufgabe nur wenig mehr als eine einfach Übung im Maschinencode ist.

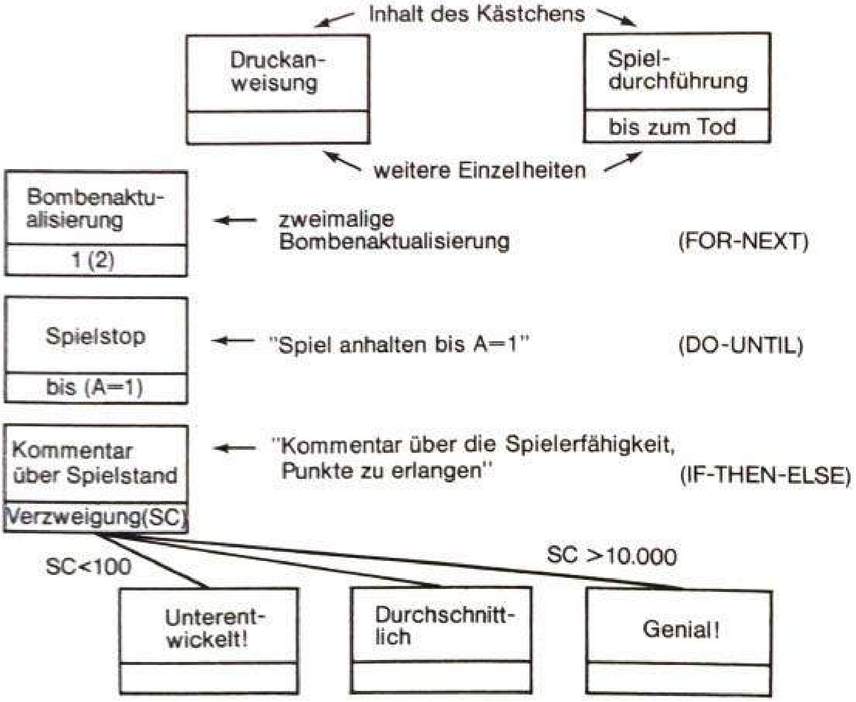
Abbildung 4.1: Baumdiagramm (Top-down-Darstellung)




Das Endergebnis erinnert an die Wurzeln eines Baumes. Nach ein wenig Übung werden Sie erkennen, wann dieser Baum ausreichend detailliert ist. Natürlich brauchen Sie diese Untergliederung nicht so weit zu treiben, bis Sie bei den eigentlichen Anweisungen selbst angelangt sind. Ich würde den Vorgang beenden, wenn die Aufgaben soweit reduziert sind, wie z.B. „Bildschirm nach links rollen" und ich erkenne, daß ich fähig bin, diese im Maschinencode zu schreiben. An diesem Punkt kommt dieses Buch ins Spiel. Es wird Ihnen zeigen, wie man solche Aufgaben, wie Rollen des Bildschirms, durchführt. Natürlich kann es nicht jede Möglichkeit abdecken, daher habe ich ein Kapitel über „Algorithmusentwurf" – Kapitel 6 – eingeschlossen. Es soll Ihren Verstand anregen, Verfahren für das Lösen spezieller Aufgaben zu ersinnen.
Jetzt, da wir eine grundlegende Einsicht in den Vorgang des Programmentwurfs erlangt haben, können wir damit beginnen, ein paar Ideen über den Entwurf von Baumdiagrammen zu entwickeln. Ist das Programm komplex, dann bedeckt das Baumdiagramm viele Seiten Papier. Das bedeutet zweierlei. Zum einen müssen wir sehr sauber und organisiert arbeiten und zum anderen müssen wir ein fehlerfreies System für die Verbindung der Kästchen untereinander haben. Ist dieses

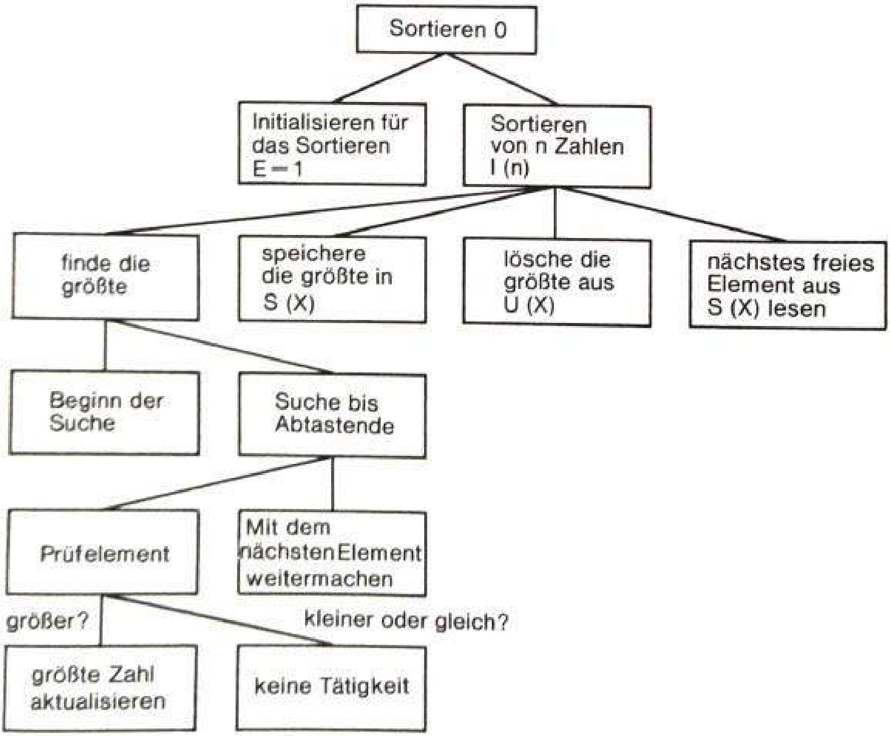
System nicht frei von Fehlern, dann wird der Baum deformiert. Eine andere Frage ist: Wie stellen wir Schleifen und Verzweigungen in einem Baumdiagramm dar? Ich werde den zweiten Punkt zuerst behandeln. Es ist ganz egal, wie Sie Schleifen und Verzweigungen in Ihren Baumdiagrammen kennzeichnen, solange Sie dieses durchgehend gleichartig tun. Die Zeiten, in denen der eine das Programm beschreibt und der andere es codiert, sind vorbei (zumindest bei Mikrocomputern). Sie schreiben dieses Programm allein. Solange Sie wissen, was Sie wollen, ist alles in Ordnung. Sie kennen wahrscheinlich die Sinnbilder, die für Ablaufpläne verwendet werden (Raute für Entscheidungen, Rechteck für Vorgänge usw.). In einem Baumdiagramm wird normalerweise nichts durch die Form der Kästchen, sondern alles durch die Zeichen innerhalb eines Kästchens ausgesagt. Ich werden Ihnen meine Darstellungsweise (die ich nicht selbst entwickelt habe) vorstellen. Es ist Ihnen freigestellt, Ihre eigene zu entwickeln, wenn Sie diese für verständlicher halten. Tun Sie dies, dann ist es ratsam, auf einem getrennten Stück Papier genau aufzuschreiben, wie Ihr System funktioniert, damit Sie nicht davon abweichen.

**Abbildung 4.2: Top-Down-Darstellung**



In meinem System wird jedes Kästchen so aufgeteilt, wie es in Abbildung 4.2 gezeigt wird. Der obere Teil enthält den Titel und die Funktion des Kästchens, während der untere Teil Einzelheiten enthält, wie das Kästchen ausgeführt wird. Für diesen Abschnitt gibt es mehrere unterschiedliche Möglichkeiten. Vielleicht möchten Sie, daß nur ein Durchlauf gemacht wird. In diesem Fall würde ich das Kästchen normalerweise leer lassen oder das griechische Symbol für Nichts 'Ø' hineinschreiben. Wir könnten aber auch eine feste Anzahl von Durchläufen vorhaben: dieses entspricht der FOR-NEXT-Schleife in BASIC. Ich zeige dies, indem ich 'I(n)' schreibe, was „n-malige Iteration" bedeutet. Vielleicht wollen Sie das Kästchen auch so oft ausgeführt haben, bis eine bestimmte Endbedingung auftritt: Dies entspricht der DO-UNTIL-Anweisung, die in einigen BASIC-Arten vorhanden ist. Sie wird mit 'UNTIL[X]' gekennzeichnet, wobei 'X' für eine Bedingung steht (z.B. SCORE (Spielstand) = 1500). Für Fälle, bei denen mehrere mögliche Ereignisse auftreten können, die von anderen Ereignissen abhängig sind, verwende ich die Schreibweise 'BRANCH ON [X]', wobei X wiederum eine

**Abbildung 4.3: Beispiel für ein Baumdiagramm**



Bedingung ist. Tritt dieser Fall auf, dann schreibe ich entlang der Verbindungslinie den Wert der Bedingung, der die Maschine diesen Weg entlang zwingt. Abbildung 4.2 soll Ihnen dies verdeutlichen.
Wir sind jetzt in der Lage, herauszufinden, wie wir alle diese Ideen entegrieren können, so daß wir ein Baumdiagramm für jeden gewünschten Zweck zeichnen können. Nehmen wir das einfache Beispiel des Sortierens einer Liste aus n Zahlen, die im Array U(X) enthalten sind, in das Array S(X). Wir verwenden dabei den folgenden Algorithmus:

(1) Suche die größte Zahl in U(X).
(2) Speichere diesen Wert im ersten leeren Element von S(X).
(3) Ersetzen Sie diese Zahl in U(X) durch eine sehr kleine Zahl (Sie löschen sie praktisch aus U(X)).
(4) Wiederholen Sie die Schritte 1 – 3, bis alle Elemente aus U(X) sehr klein sind.

Dies kann in dem in Abbildung 4.3 gezeigten Baumdiagramm dargestellt werden. Denken Sie daran, daß wir das Prinzip des Entwurfs von Baumdiagrammen demonstrieren wollen und deshalb dieses Beispiel in BASIC erläutern werden. Bevor ich es Ihnen erkläre, versuchen Sie, zu verstehen, wie es arbeitet. Das tatsächliche Sortierkonzept für eine Reihe von Zahlen ist ziemlich komplex, aber die Anwendung eines Baumdiagramms teilt es in leicht zu behandelnde Stücke auf.
Sehen wir uns die Abbildung 4.3 an. Das Diagramm beginnt mit „Zahlen sortieren". Das ist eine logische Möglichkeit, anzufangen. Die '0' im Kästchen ist nicht unbedingt erforderlich, sie steht nur da, um Ihnen zu zeigen, was das Kästchen bedeutet. Diese Aufgabe wird in zwei Teilaufgaben zerlegt: die Vorbereitung für die Aufgabe und deren Ausführung. 'E = 1' im linken Kästchen bedeutet – Zuweisung des Wertes 1 zur Variablen E. Das ist offensichtlich keine Anweisung. Das rechte Kästchen enthält unser Iterationssymbol 'I(n)'. Ein Blick auf den Algorithmus wird Sie daran erinnern, daß wir den Vorgang des Herausfindens der größten Zahl und ihre Aufzeichnung so oft durchführen müssen, wie Zahlen in der Liste vorhanden sind. Das Symbol teilt uns mit, daß die Ausführung der untergeordneten Kästchen n-mal erfolgen muß. In der nächsten Kästchenebene ist es unmöglich, aus einem der Kästchen auf den Zweck des gesamten Diagrammes zu schließen. Das ist ein deutlicher Hinweis darauf, daß unsere Vereinfachung des Problems funktioniert. Die 4 Kästchen umfassen 4 Schritte, die für die jeweils gefundene größte Zahl wiederholt werden müssen. Für eins dieser 4 Kästchen ist es nicht eindeutig klar, wie wir es programmieren sollen. Deshalb wird dieses Kästchen weiter untergliedert. Die Funktionen dieser Verzweigung sollten Ihnen klar sein: Beachten Sie die bedingte Verzweigung mit der entsprechenden Kennzeichnung an jedem Ast.

Jetzt können wir dieses Diagramm programmieren. Da dieses später sachlich diskutiert wird, wird jetzt nur ein kurzes Beispiel gegeben. Aus Gründen der Einfachheit programmieren wir in BASIC. Das hier verwendete Prinzip ist als „Abarbeiten eines Strukturbaumes" bekannt. D.h. wir beginnen oben und arbeiten uns am äußersten linken Ast herab, bis wir an einem toten Ende (oder „Blatt") ankommen. Dann kehren wir um, gehen zurück und nehmen den nächsten möglichen Weg an der nächsten Verzweigung, der noch nicht erforscht worden ist. In dieser Weise können wir den gesamten Baum durchlaufen.
Wenn wir oben beginnen, dann treffen wir zuerst auf „Nummern sortieren". Deshalb schreiben wir folgendes in die erste Zeile:

```
10 REM *** NUMMERN SORTIEREN ***
```

Als nächstes kommen wir zu E = 1, daraus ergibt sich die zweite Zeile.

```
20 LET E=1
```

Das folgende Kästchen ist „Sortiere n Zahlen", also wird unsere dritte Zeile

```
30 GOSUB 1000
```

was bedeutet, daß das Unterprogramm, das in Zeile 1000 beginnt, diese Funktion durchführt. Es ist angebracht, daß wir dies deutlich machen, und deshalb lautet Zeile 1000

```
1000  REM *** SORTIERE N ZAHLEN ***
```

Wir gehen den Baum weiter entlang und kommen bei „Suche die größte" an. Das teilt sich selbst auf, und deshalb schreiben wir:

```
1010  GOSUB 2000
```

und

```
2000  REM *** SUCHE DIE GRÖSSTE ***
```

Das bringt uns zu „Suche beginnen" – endlich ein Blatt! – Das Ergebnis:

```
2010  LET L=1
```

Als nächstes führt unsere Reise uns zu „Suche", und wir erhalten die Anweisung, fortzufahren bis L = 101. Da dieses kein Blatt ist, schreiben wir:

```
2020  GUSUB 3000
```

gefolgt von

```
3000  REM *** SUCHE ***
```

Jetzt kommen wir zum „Test", der eine bedingte Verzweigung ist. Unter Berücksichtigung aller Kästchen in der Reihenfolge schreiben wir:

```
3010 IF N(L)>LARGEST THEN LET LARGEST=N(L):LET P=1
```

Da im nächsten Kästchen nichts passiert, können wir es genauso gut ignorieren. Unsere Wanderung führt uns nun zu „nächstes L", was ein Blatt ist, wir schreiben:

```
3020  LET L=L+1
```

Das war das letzte Kästchen in der Aufgliederung von „Suche", und so können wir schreiben:

```
3030  RETURN
```

Wir befinden uns jetzt wieder in Zeile 2020, in die wir ein GOSUB gesetzt haben. Es gab da jedoch eine Anweisung, diese Operation solange zu wiederholen, bis L=101, deshalb schreiben wir:

```
2030  IF L<>101 THEN GOTO 2020
```

Wir gehen jetzt wieder den Baum hinauf und beenden die Subroutine „Suche die größte", indem wir folgendes schreiben:

```
2040  RETURN
```

Wir sind zurück in der Subroutine aus Zeile 1000 und stellen fest, daß wir drei Blätter haben, deshalb schreiben wir:

```
1020  S(E)=LARGEST
1030  N(O)=-9999
1040  E=E+1 (das optionale „LET" habe ich weggelassen)
```

Das ist wieder ein Blatt, deshalb beenden wir dieses Kästchen mit

```
1050 RETURN
```

Wir kehren jetzt zu den ersten Zeilen zurück, die wir geschrieben haben, und fügen

```
40 END
```

hinzu, um den Computer zu stoppen.

Das Programm ist jetzt vollständig und bereit zur Überprüfung. Sie werden bemerkt haben, daß das Verzweigungsverfahren das Programm praktisch für uns schrieb. Wir mußten nur ein wenig nachdenken.

Es sollte jetzt klar sein, daß ein vollständiges Programm ein sehr großes Baumdiagramm erfordert. Man kann dieses nur sehr schwer auf einem einzelnen Blatt Papier darstellen. Die Antwort darauf ist: Verwenden sie möglichst großformatiges Papier (z. B. DIN A 4), die alle mit einer Kennummer versehen sind. Diese könnte aus zwei Buchstaben für den Spielenamen und der laufenden Nummer des Blattes bestehen.

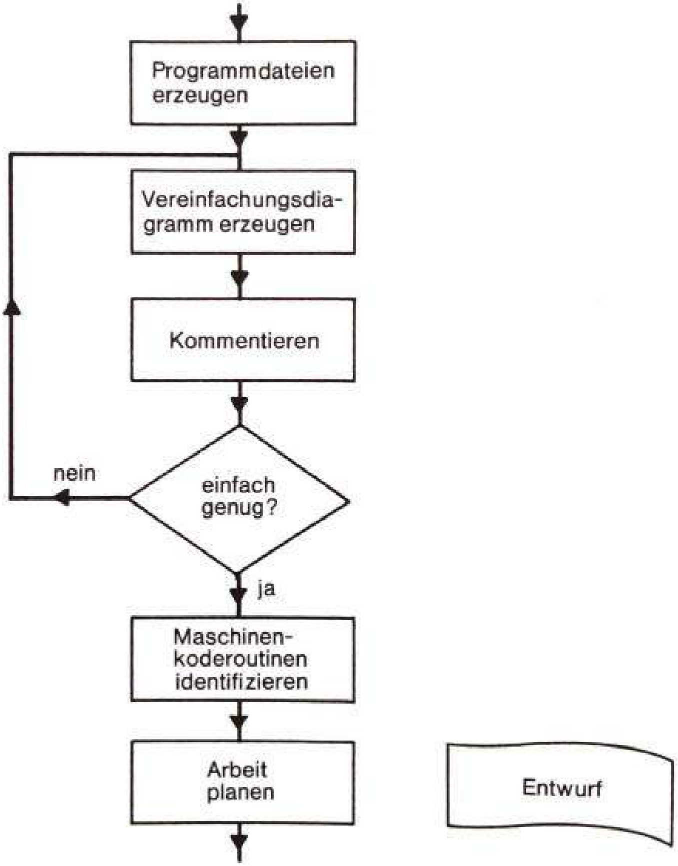
Die Kennummer soll Ihnen bei der Identifizierung des Spiels, zu dem das Diagramm gehört, helfen. Z. B. für Space Invaders könnten wir SI1, SI2, SI3 usw. haben. Die Bezugsnummer des Diagramms sollte deutlich in ein Kästchen oben auf die Seite geschrieben werden. Erstellen Sie auf einem anderen Blatt Papier ein Inhaltsverzeichnis dieser Bezugsnummern und des Zwecks der Diagramme. Das ermöglicht einen schnellen und einfachen Querverweis. Zur Identifizierung jedes einzelnen Kästchens einer Seite verwenden Sie den Seitencode mit vorangestelltem Buchstaben „b" (für box = Kästchen). Außerdem ist noch ein weiterer Code zur Unterscheidung aller Kästchen einer Seite anzuhängen. Wenn Sie noch Lust haben, dann stellen Sie ein weiteres Verzeichnis auf, in dem alle diese Codes aufgeführt sind. Diese Papierarbeit hilft Ihnen in späteren Stadien der Programmerstellung sehr.

Haben wir erst einmal ein Baumdiagramm erstellt, dann können wir fast mit dem Codieren beginnen. Wir können klar erkennen, was wo erforderlich ist. Weiterhin können wir, wenn wir das Diagramm genau ansehen, Wiederholungen im Code lokalisieren. Diese sind natürlich ideale Ansätze für Unterprogramme. Das Diagramm tut aber noch mehr für uns: Es zeigt uns, wie die Programmteile zu verbinden sind und den natürlichen Weg, wie jeder Abschnitt geprüft werden kann, wenn er zur Integration in den Rest des Programms verfügbar wird.

Der effizient arbeitende Programmierer sollte für das gesamte Vorhaben einen Zeitplan erstellen. Damit kann er jederzeit erkennen, wo er sich genau befindet (und ob er mit seiner Arbeit im Plan ist). Lassen Sie sich für jede einzelne Realisierungsstufe viel Zeit, da der Maschinencode niemals ohne Fehler geschrieben wird.

Ein geübter BASIC-Programmierer kann ein vollständiges 10K-Programm in zwei Tagen schreiben und zum Laufen bringen, aber ein Maschinencode-Programmierer muß sich schon beeilen, um 1K benutzbarer Codes an einem Tag zu schreiben.
Die Stufen für die Erstellung eines Plans variieren von Programmierer zu Programmierer. Einige Programmierer benötigen nicht viel Organisation. Bei einem tiefergehenden Programm kann das aber bereits eine geistige Höchstleistung bedeuten. Die erste Stufe ist die Durchsicht des Diagramms und die Identifizierung aller Routinen. Danach sind, in Abhängigkeit von dem Anteil der Abläufe, die im Maschinencode sein sollen, alle Maschinencode-Routinen herauszusuchen.

**Abbildung 4.4: Entwurf**



Tragen Sie alle diese Routinen einschl. Ihrer Verweise auf das Diagramm in einer Liste zusammen. Anschließend arbeiten Sie das Diagramm durch, schätzen ab, wie lange Sie für das Codieren und die Prüfung jeder Routine benötigen (Erfahrungswerte) und erstellen Sie daraus Ihren Zeitplan. Es ist gut, sich anzugewöhnen, den Zeitplan strikt einzuhalten, deshalb seien Sie großzügig mit den Schätzwerten.
Bevor wir zum nächsten Kapitel übergehen, versuchen Sie, einige BASIC-Programme nach dem Top-down-Verfahren zu schreiben. Dadurch erhalten Sie eine Chance, den vollen Wert des Verfahrens wirklich schätzen zu lernen. Es ist genau das, was als „strukturierte" Programmierung bekannt ist, denn das Diagramm zeigt die Programmstruktur. Um rationeller zu arbeiten und weil die Darstellung dadurch deutlicher wird, verwende ich eine Zeichenschablone, die alle Sinnbilder für Ablaufpläne und Baumdiagramme enthält. Außerdem verwende ich einen Bleistift, um Ideen zu skizzieren.
Habe ich den Prototyp eines Programmes entworfen, dann übertrage ich es auf ein neues, sauberes Blatt Papier. Bei Verwendung einer Schablone erhalten Sie gleichmäßige Kästchen mit rechteckigen Seiten. Sie können eine solche Schablone kaufen oder aus Karton ausschneiden.
Noch eine letzte Bemerkung zu diesem Kapitel. Achten Sie darauf, daß Sie beim Zeichnen dieses Diagramms die Fundamente des Spiels legen. Sie sollten wirklich sicher bauen. Nehmen Sie sich also Zeit und überprüfen Sie das Baumdiagramm lieber zweimal, wenn es fertig ist.

# KAPITEL 5

# KODIEREN DES PROGRAMMES

Jetzt haben Sie das Fundament gelegt und wir können endlich mit dem Schreiben des Programmes beginnen. Machen wir zuerst eine Inventur dessen, was wir bereits getan haben. Wir besitzen eine detaillierte Programmbeschreibung und haben Baumdiagramme erstellt, die uns genau erklären, wie wir das Programm zu schreiben haben. Weiterhin wissen wir, welche Routinen im Maschinencode zu schreiben sind. Deshalb sind wir jetzt in der Lage, mit dem Schreiben des Programmes zu beginnen.

Ein wesentlicher Teil dieses Buches wird von einem Abschnitt mit „allgemeinen Programmroutinen" (Kapitel 9 und 10) eingenommen. Er enthält eine große Auswahl allgemeiner Programme, die Sie wahrscheinlich gerne verwenden werden. Diese reichen vom Ausfüllen von Speicherbereichen mit einem Code bis zum Rollen des Bildschirms nach links und rechts und in andere Richtungen. Jede Routine wird hinsichtlich ihres Gegenstandes und Stils besprochen, so daß sie leicht anzupassen sind. Dieses Kapitel erläutert, wie Sie diese Routinen bei minimaler Anpaßarbeit maximal ausnutzen können. Glauben Sie aber nicht, daß dieses Buch das Schreiben des Spiels für Sie übernimmt: Es wird immer noch ein paar Routinen geben, die Sie selber konstruieren müssen. Kapitel 6 ist diesem Aspekt gewidmet.

Der aufmerksame Leser wird bereits bemerkt haben, daß es zwei grundsätzlich verschiedene Arten von Routinen in unserem Programm geben wird. Zum einen gibt es die rein funktionellen Routinen, die in jedem Blatt des Baumdiagramms auftauchen: Diese Routinen führen beachtliche Aufgaben durch. Die zweite Art von Routinen strukturieren das Programm. Sie verbinden einzelne Routinen miteinander und führen Schleifenoperationen zwischen den Routinen durch. Die letzteren bilden die Knoten (d. h. Verästelungen) des Baums. Das Schreiben dieser Verbindungsroutinen wird in Kapitel 8 besprochen. Das Kapitel, welches Sie gerade lesen, befaßt sich nur mit den rein funktionellen Routinen.

Ich setze voraus, daß Sie bereits ein recht gutes Wissen über den Maschinencode besitzen. Daß Sie also verstehen, wie man Register bearbeitet und einfache Additions- und Subtraktionsoperationen durchführt, womit sich Bücher über den 6510- und 6502-Code ausführlich beschäftigen. Jedoch in bezug auf das Schreiben von etwas wirklich Nützlichem, lassen diese Bücher (von einigen Ausnahmen abgesehen) einen meistens im Dunkeln. Das geschieht deswegen, weil es sehr schwer ist, im Maschinencode objektiv zu sein. Die Sprache eignet sich sehr gut für die Konstruktion von kurzen, schnellen, sich wiederholenden Aufgaben, während das Schreiben eines komplexen Programmes eine entmutigende Aufgabe ist.

Durch die Einführung des Baumdiagramms haben wir jedoch unsere Aufgabe auf eine Reihe von sehr einfachen Unteraufgaben reduziert, die sich sehr gut für eine objektive Programmierart eignen. Ist dies bei Ihren Diagrammen nicht der Fall, dann haben Sie einfach den Baum noch nicht genügend detailliert.
Eine entscheidende Anweisung im Maschinencode ist die JSR/RTS-Kombination. Diese beiden Anweisungen geben Ihnen die Möglichkeit, Subroutinen genauso wie in BASIC aufzurufen. Der einzige Unterschied ist, daß wir viel häufiger verschachteln können, als in BASIC. Der 6510 ermöglicht uns bis zu 128 JSR-Aufrufen ohne ein einziges RTS. Dadurch können wir bis zu 127 Subroutinen innerhalb einer Subroutine aufrufen. Mit einem kurzen Vorgriff auf das nächste Kapitel kann man sagen, daß die „Verbindungsroutinen" durch ein Aufrufen der untergeordneten Routinen mit JSRs arbeiten. Wenn Sie sich noch nicht vollständig klar über den Zweck der JSRs sind, dann lesen Sie sich erst einmal die entsprechenden Abschnitte in Ihrem Handbuch durch.

## ÜBERGABE VON PARAMETERN

Die Übergabe von Parametern ist ein ungeheuer wichtiges Konzept. Parameter sind Informationsteile, die eine Routine benötigt, um ihre Aufgabe zu erfüllen. Die Parameter einer Print-Anweisung in BASIC sind die Variablen und der Text, den Sie drucken möchten. Folglich benötigen viele Routinen aus dem Maschinencode irgendwelche Parameter. Einige kommen jedoch ohne sie aus. Um ein Beispiel zu geben, betrachten wir eine Routine, die einen Speicherabschnitt mit einem bestimmten Zeichen füllt. Zur Durchführung dieser Funktion muß die Routine wissen, wo sie mit dem Füllen beginnen und wo sie aufhören muß und welche Zeichen zu füllen sind. In BASIC gibt es nur einen Weg der Parameterübergabe. Sie weist die zu übergebenden Werte einer Variablen zu, die die Routine abfragen muß. Im Maschinencode gibt es zwei unterschiedliche Verfahren. Entweder können die erforderlichen Werte in vorbestimmte Speicherstellen abgelegt werden, so daß die Routine sie herausgreifen kann, oder die Daten können dem Stapelspeicher übergeben werden. Für unsere Zwecke werden wir das erste Verfahren verwenden, da es einfacher durchzuführen ist.
Lassen Sie mich dieses Verfahren einmal demonstrieren. Nehmen wir einmal an, daß wir eine Auffüllroutine verwenden wollen und deshalb zwei Adressen und einen Code an die Routine weitergeben müssen. Solange die Parameter in der gleichen Weise gesendet und empfangen werden, treten keine Probleme auf. Wir können die beiden Adressen in den Stellen 251 – 252 und 253 – 254 als ein Paar 16-stelliger Zahlen speichern. Da der Code, mit dem der Bereich gefullt wird, nur ein einzelnes Byte umfaßt, können wir es in das X-Register übertragen. (Diese Auswahl hat nichts zu bedeuten, man könnte genauso das Y-Register nehmen.)

Solange die Auffüllroutine dies weiß, ergeben sich keine Probleme. Daraus folgt eine wichtige Tatsache. Soll eine Routine im gesamten Programm mit nur einem Parametersatz verwendet werden, dann wäre es sinnvoller, ihre Übergabe vollständig zu vergessen und sie einfach als Teil der Routine selbst zu initialisieren. Sie müssen selbst herausfinden, wann eine solche Situation auftritt, und dann entsprechend handeln.
Die gleiche Argumentation gilt auch für die Übergabe der Parameter aus der Routine zurück in die aufrufende Routine und danach in die nächste Routine. Dieses wird weiter unten in diesem Kapitel sowie im Kapitel 8 detaillierter besprochen. Jetzt werden wir uns aber erst einmal ein paar Routinen ansehen, die wir codieren möchten. Dabei wollen wir uns sowohl die Parameterübergabe, als auch die tatsächliche Funktion der Routine ansehen. Natürlich gibt es ein paar Routinen, die keine Parameter benötigen.

## ZUSAMMENSTELLUNG VON ROUTINEN

Falls Sie bereit sind, können wir mit dem Zusammenstellen von einigen Routinen beginnen. Dabei wollen wir nicht einfach nur Papier bekritzeln. Sie müssen schon ein bißchen darüber nachdenken, was Sie tun. Das System, das ich hier beschreibe, soll Ihnen nur die richtige Richtung zeigen, um Sie in Gang zu bringen, später brauchen Sie sich dann nicht mehr strikt daran halten. Es ist ganz besonders wichtig, in diesem Stadium nicht zu schnell zu arbeiten – Sie werden sonst keinen Erfolg haben und nur enttäuscht sein. Das Arbeiten im Maschinencode ist ein langsamer und arbeitsreicher Vorgang. Seien Sie darauf gefaßt, daß Sie Fehler machen. Es wird für Sie ein echter Test Ihres Durchhaltevermögens werden.
Im ersten Stadium der Zusammenstellung sollten Sie sich die zu codierende Routine genau ansehen. Sind Sie sich ihrer Funktion vollständig bewußt? Ist sie einfach genug, um codiert zu werden, oder haben Sie den Baum nicht bis zum erforderlichen Ausmaß verzweigt? Sie müssen bis zur letzten Einzelheit wissen, was Sie erreichen wollen.
Weiterhin müssen Sie sich bewußt sein, welche Parameter für die Routine erforderlich sind.
Die nächste Stufe ist das Suchen und Herausfinden der entsprechenden Hilfsroutine (wenn es eine gibt). Vielleicht finden Sie eine Hilfsroutine, die der zu codierenden Routine mehr oder weniger ähnlich ist. Meistens ist die zu codierende Routine eine Verbindung von mindestens zwei Hilfsroutinen. Ist keine wirklich ähnliche Routine vorhanden, dann werden Sie die Routine wohl oder übel ohne Vorgabe selbst schreiben müssen. Legen Sie sie erst einmal zur Seite und machen Sie mit der nächsten Routine weiter. Das Schreiben ohne Vorgabe wird im nächsten Kapitel behandelt.

Damit dies nicht zu langsam geht, wäre es ratsam, zunächst alle Routinen gut kennenzulernen, so daß Sie mit ihnen vertraut sind, wenn Sie sie benutzen müssen. Aber lassen Sie es nicht dabei bewenden. Stellen Sie Ihre eigenen Routinen auf und dokumentieren Sie sie gut. Fangen Sie dann an, sie im Geiste zu mischen. Sie werden überrascht sein, welche neuen Routinen Ihnen dabei einfallen werden.

Studieren Sie jede Routine und legen Sie die Parameter fest, die sie benötigt. Diese können zwei Klassen zugeordnet werden: eingehende und ausgehende. Bei eingehenden Daten müssen Sie die herausfinden, die konstant bleiben. Diese Information kann genauso gut von der Routine selbst und nicht von der Abrufroutine eingegeben werden. Diese Informationen sollten sorgfältig in den Programmdateien aufgezeichnet werden, damit sie bei Bedarf verfügbar sind.

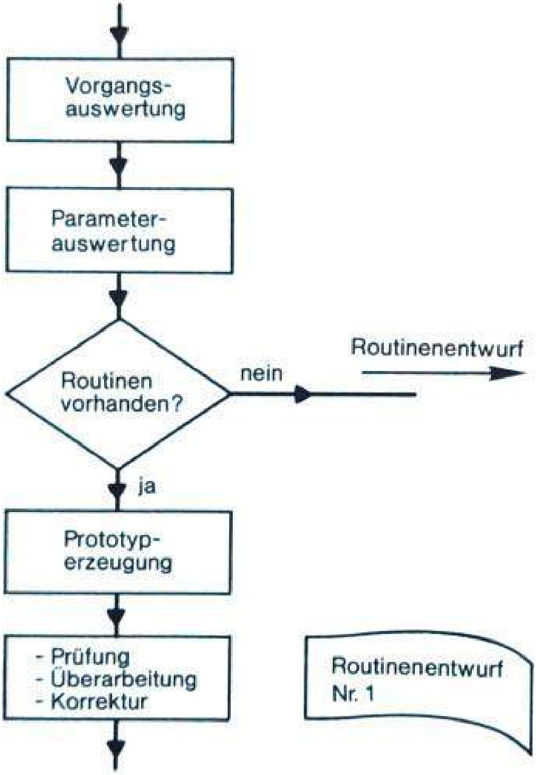
Ich muß an dieser Stelle auf die Gefahr von Parameterkollisionen hinweisen. Diese treten auf, wenn mehrere Routinen den gleichen Speicherbereich für unterschiedliche Parameter verwenden. Dies führt zu Datenverlust und einer fehlerhaften Eingabe bei ihrem Austausch. Meistens spielen diese Kollisionen keine Rolle, da die in diesen Stellen enthaltenen Informationen nicht weiter benötigt werden, manchmal verursachen sie aber eine Katastrophe und es kann sehr schwierig sein, den Fehler zu finden. Solange die Routinen alleine arbeiten, funktionieren sie einwandfrei, sobald sie aber zusammen ausgeführt werden, treten Fehler auf. Das kann sehr frustrierend sein. Um dieses zu verhindern, müssen Sie eine detaillierte Aufzeichnung erstellen, welche Speicherstellen wo verwendet werden.

Wenn Sie die Anweisungen bis jetzt befolgt haben, sind Sie in der Lage, zumindest einige Routinen selbst zu schreiben. Egal, ob Sie einen Assembler besitzen oder nicht, Sie brauchen von allen Routinen, die Sie schreiben, eine Kopie, damit sie sofort verfügbar sind. Ich würde das Kopieren der Routinen zu einer festen Regel machen, damit alles vollständig klar ist. Sie können sich keine Unklarheiten leisten. Die erste Version ist der Prototyp – sorgen Sie dafür, daß er funktioniert. Es spielt keine Rolle, wenn er ein wenig unrationell ist. In diesem Stadium sollten Sie keinen, der reichlich in Maschinencode-Programmen enthaltenen, cleveren Tricks verwenden. Halten Sie ihn einfach. Versuchen Sie, JMP-Befehle innerhalb einer Routine zu vermeiden, da dadurch der Code nicht mehr an eine andere Stelle angeordnet werden kann. Es darf keinen Aufruf für einen Sprung von mehr als 127 Byte geben, da dies der Grenzwert für relative Verzweigungen ist. Verwenden Sie eine bedingte Verzweigung als unbedingte Verzweigung, dann erzwingen Sie die Verzweigung, indem Sie die Marke setzen, die diese Verzweigung regelt. Ein Beispiel dafür ist eine CLC-Anweisung vor eine BCC-Anweisung zu setzen. Wenn Sie von Hand assemblieren, dann hüten Sie sich vor der Berechnung dieser Vektoren. Sie sind eine häufige Fehlerquelle.

Haben Sie den Prototyp Ihrer Routine erst einmal assembliert, geprüft und korrigiert, dann wird er in der nächsten Stufe verbessert. Vielleicht gefällt er Ihnen so wie er ist. Vielleicht haben Sie aber ein paar Ansätze für Verbesserungen bemerkt.

die Sie hinzufügen könnten. Ist das der Fall, dann machen Sie weiter und ändern Sie Ihre Routine. Vergessen Sie aber nicht diese nach der Änderung erneut zu testen.

**Abbildung 5.1: Routinenentwurf Nr. 1**

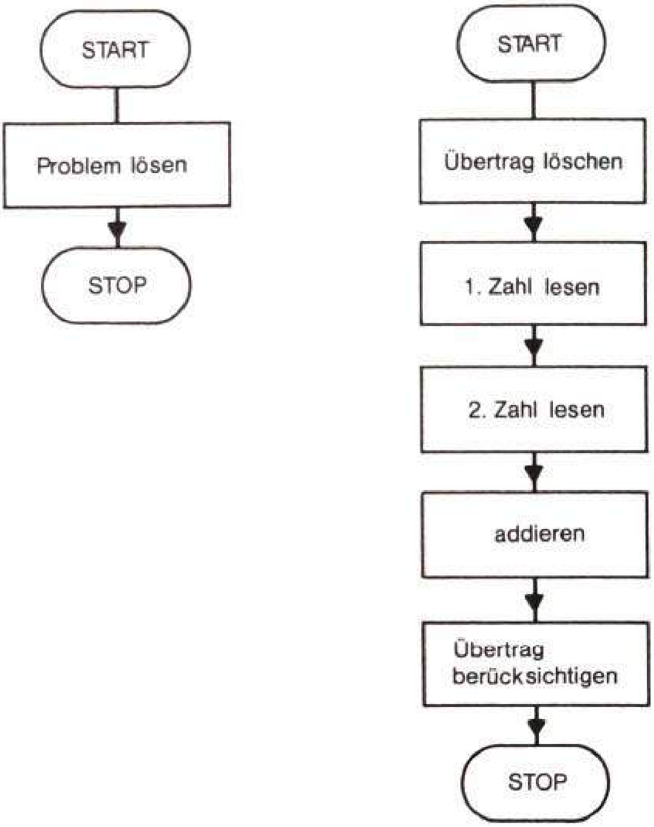


# KAPITEL 6

# SCHREIBEN EIGENER ROUTINEN

Nicht alle Routinen, die Sie haben möchten, können für Sie geschrieben werden. Sie können entweder versuchen, sie selbst zu schreiben, oder vom Maschinencode zurückschrecken und sie in BASIC schreiben. Machen Sie zumindest einen Versuch im Maschinencode: Auch wenn es kläglich daneben geht, haben Sie immer noch wertvolle Erfahrung gewonnen.
Zuerst sollten Sie überlegen, ob Sie das, was Sie erreichen wollen, in gewöhnlichen Worten ausdrücken können. Nehmen wir an, Sie wollen ein Schachprogramm schreiben. Es ist schwierig genug, den Ablauf deutsch zu beschreiben, ganz zu schweigen davon, es in den Maschinencode zu übertragen. Ist dies der Fall, dann haben Sie Probleme mit dem Algorithmus (siehe hierzu den Abschnitt über Algorithmusentwurf weiter unten in diesem Kapitel). Erwarten Sie aber keinen sofortigen Erfolg, die Inspiration läßt manchmal auf sich warten.
Nehmen wir also an, Sie haben Ihren Algorithmus ausgearbeitet. Dann können wir jetzt daran gehen, ihn in eine funktionierende Routine im Maschinencode umzuwandeln. Zeichnen Sie zuerst einen Ablaufplan des Algorithmus. Zur Planung von Programmen als Ganzes ist ein Ablaufplan ein schlechtes Hilfsmittel, aber für die Festlegung eines Algorithmus ist es sehr hilfreich. Versuchen Sie, jedes Kästchen so einfach wie möglich zu machen. Sie brauchen aber nicht bis zu den Anweisungen selbst zu gehen. Was wir brauchen, ist irgend etwas, das zwischen den beiden Flußdiagrammen von Abb. 6.1, die zwei Zahlen addiert. Falls erforderlich, verdeutlichen Sie die Programmflußlinien mit Pfeilen. Kann die Programmsteuerung verschiedene Wege nehmen, dann sollten diese auf ein Mindestmaß beschränkt werden. Das Diagramm würde sonst überladen. Eine Hauptursache für Störungen im Maschinencode (und anderen Computersprachen) ist eine zu komplizierte Verzweigungsstruktur. Diese führt zu unvorhergesehenen logischen Fehlern und hat einen sehr frustrierten Programmierer als Ergebnis. Falls erforderlich, sollte man den Ablaufplan mehrere Male neu zeichnen, und dabei sowohl die Struktur, als auch den Befehlsinhalt vereinfachen.
Zum jetzigen Zeitpunkt sollten Sie genau wissen, welche Parameter an das und von dem Programm zu übergeben sind. Schreiben Sie diese oben und unten auf ein Stück Papier. Das Flußdiagramm sollte den Hauptteil oder „das Rückgrat" der Routine verdeutlichen. Machen Sie sich keine Gedanken über andere Kleinigkeiten, schreiben Sie nur den Code auf das Blatt, von dem Sie glauben, daß er diese Aufgabe erfüllt. Sollten Sie sich Ihrer Sache nicht sicher sein, dann tippen Sie ihn ein und überprüfen Sie ihn. Danach können Sie damit beginnen, die anderen „Arme" des Ablaufplanes hinzuzufügen. Machen Sie nach jeder Stufe eine Pause und prüfen Sie, was Sie geschrieben haben. Machen Sie nicht alles in einem

**Abb. 6.1: Zu geringe und übermäßige Aufgliederung eines Ablaufplans**




Durchlauf. Haben Sie die Routine fertig geschrieben, dann sehen Sie sich die Parameter oben und unten auf dem Blatt an und fügen Sie den Übergabecode hinzu. Damit haben Sie den Prototyp fertiggestellt. Funktioniert er, dann ist es gut, aber wahrscheinlich haben Sie bereits Pläne für eine zweite, bessere Version. Die folgenden Diagramme und Codes zeigen dieses Verfahren in der Anwendung.

## ENTWICKLUNG EINER ROUTINE FÜR EIN SPIRALFÖRMIGES AUFFÜLLEN DES BILDSCHIRMES

Das Auffüllen des Bildschirms von oben nach unten ist ein Kinderspiel. Von innen nach außen ist es nicht so einfach. So etwas im Maschinencode zu versuchen, erscheint bestimmt zuerst einschüchternd. Die Lösung ist jedoch gar nicht so schwierig. Zeichnen Sie einfach, unter Verwendung der Rechteckauffüllroutine (siehe Auffüllen des Speichers, Kapitel 9), immer größere Rechtecke um den Mittelpunkt des Bildschirms herum. Als Ergebnis erhalten Sie einen Bildschirm, der sich von innen nach außen füllt.
Sehen wir uns die Rechteckauffüllung an, dann erkennen wir, daß sie mehrere Parameter benötigt. Als erstes braucht sie die Adresse der oberen linken Ecke des Rechtecks, danach die Breite und Höhe und schließlich den Code, mit dem sie den Bereich füllen muß. Diese werden in die folgenden Stellen gesetzt:

| | |
|---|---|
| 251 & 252 | niederwertige und höherwertige Adresse der Ecke |
| 253 | Breite |
| 254 | Höhe |
| 2 | Auffüllcode |

Die Bildschirmbreite beträgt 40 Zeichen und die Höhe 5 Zeichen. Erhöhen wir zur Erzeugung des nächsten Rechtecks die Länge jeder Seite um 1, dann wird das Ergebnis asymmetrisch. Wir müssen um 2 erhöhen. Das bedeutet, daß die Anfangshöhe 1 beträgt (damit wir 25 und nicht 24 erhalten). Daraus folgt, daß 12 Vergrößerungen benötigt werden, um den Bildschirm zu füllen. Das führt insgesamt zu 13 Rechtecken. Eine einfache Rechnung ergibt die Anfangsbreite:

12 x 2 (12 Vergrößerungen) + Breite = 40
Breite = 40 - 12 x 2
= 16

Die Anfangsecke erhalten wir mit einer weiteren Rechnung. Nach jeder Vergrößerung bewegt sich die Ecke nach links und nach oben – diagonal nach Nordwest. Wir wissen, daß die Endposition nach 12 Bewegungen 1024 sein muß, also gilt folgendes:

ADRESSE -12 x -41 (41 ist der Sektor) = 1024
somit wird die ADRESSE = 1024 + 12 x 41
= 1516

Jetzt können wir den ersten Ablaufplan zeichnen (Abb. 6.2 und 6.3).

**Abbildung 6.2:**
**Vorläufiger Ablaufplan**

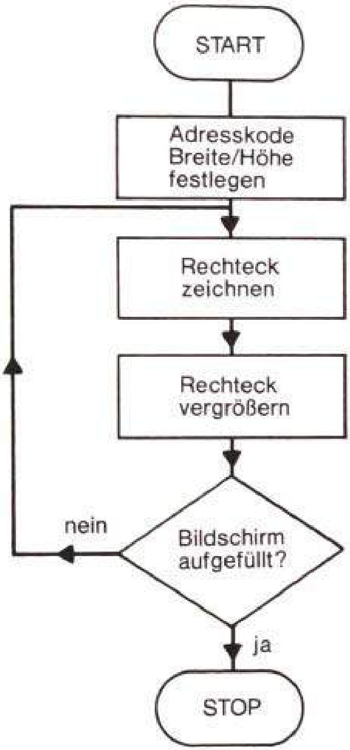
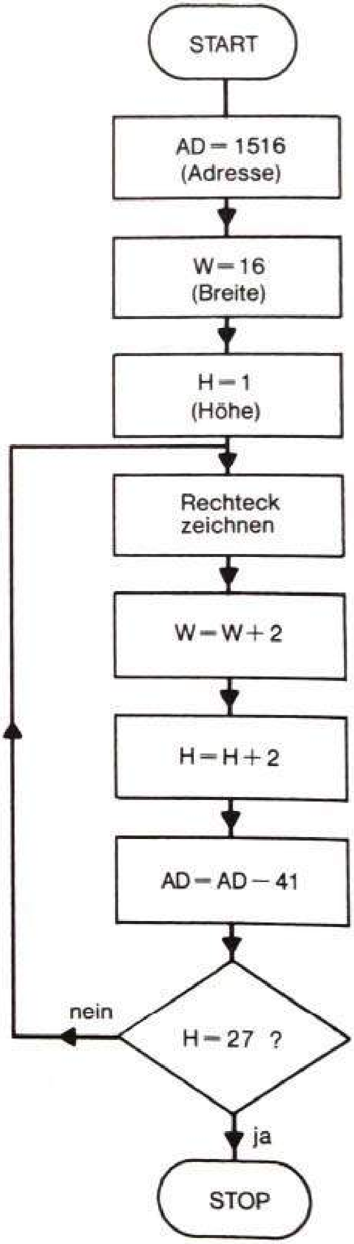



**Abbildung 6.3:**
**Codierfähiger Ablaufplan**



**Spiralförmige Auffüllung**

```
:LL0    LDA #236
        STA  ADDR
        LDA #5
        STA  ADDR+1
        LDA #16
        STA  WIDTH
        LDA #1
        STA  HEIGHT
:LL1    LDA  ADDR
        STA  251
        LDR  ADDR+1
        STA  252
        LDA  WIDTH
        STA  253
        LDA  HEIGHT
        STA  254
        JSR  FILL
        INC  WIDTH
        INC  WIDTH
        INC  HEIGHT
        INC  HEIGHT
        LDA  ADDR
        SBC #41
        STA  ADDR
        LDA  ADDR+1
        SBC #0
        STA  ADDR+1
        LDA  HEIGHT
        CMP #27
        BNE  LL1
        RTS
```

Diese Routine erzeugt eine Auffüllung des Bildschirmes von der Mitte nach außen. Sicherlich ist sie aufwendig, aber sie funktioniert. Weitere Verfeinerungen sollten erst gemacht werden, wenn der Prototyp der Routine fertig ist. Sie möchten eine Verzögerung erreichen, um das Geschehen zu verlangsamen – es ist wie immer: Aus Schaden wird man klug.

# ALGORITHMUSENTWURF

Früher oder später werden Sie auf diese Seite stoßen. Natürlich kann dieses Buch nicht jede Eventualität abdecken: Würde jede Routine, die jemals geschrieben werden könnte, hier aufgelistet, dann wäre der gesamte Zauber der Computerwelt zerbrochen. Computer bieten praktisch unendlich viele Möglichkeiten. Und genau hier setzt dieser Abschnitt an. Er dient dazu, Ihre Sinne anzuregen, so daß Sie in der Lage sein werden, Probleme zu lösen und neue Routinen zu erzeugen, um Funktionen auszuführen, von denen Sie bisher nur geträumt haben. Das Problem kann einfach oder kompliziert sein. Oder, schlimmer noch, es kann aufreizend leicht, aber nur schwer zu lösen sein. Dieses Kapitel behandelt den Entwurf von Algorithmen. Das Wort Algorithmus beinhaltet einen Weg, etwas zu tun. Dieses Kapitel soll Ihnen zeigen, wie Sie Ihre eigenen Algorithmen erzeugen können.

Der erste Schritt (so einfach er auch erscheinen mag) ist die präzise Definition des Problems. Schließen Sie jede Einzelheit der Aufgabe ein. Betrachten Sie das Problem und versuchen Sie, es in kleinere Aufgaben zu zerlegen. In dem Abschnitt über den Programmentwurf wurden Sie bereits gebeten, dieses zu tun, aber hier suchen wir nach subtileren Verbindungen der Teilaufgaben. Sie haben uns die Aufgabe soweit wie möglich vereinfacht, aber jetzt müssen wir daran denken, daß wir es mit zwei Routinen zu tun haben, die vielleicht parallel zueinander arbeiten. Betrachten wir folgendes einfaches Beispiel. Ein zufällig herumwanderndes Objekt ist innerhalb eines Quadrates zu halten. Das Problem kann nicht dadurch gelöst werden, daß man festlegt, daß das Teilchen sich niemals um mehr als einen festen Abstand von einem zentralen Punkt entfernen darf. Das würde einen Kreis beschreiben. Sie können keinen Algorithmus finden, der auf dem Abstand aufbaut. Die Antwort lautet, man muß die senkrechten und waagerechten Abstände getrennt betrachten. Erwarten Sie nicht, daß Ihnen eine solche Antwort sofort einfällt. Sie ist nur ein Schritt in die richtige Richtung. Versuchen Sie nicht, einen Algorithmus für sich alleine zu lösen. Betrachten Sie mehrere auf einmal, da Sie wahrscheinlich herausfinden, daß der eine dem anderen hilft. Die Fähigkeit, klar und unvoreingenommen zu denken, sind wichtige Eigenschaften, die Sie sich beim Algorithmusentwurf aneignen.

Läuft diese erste Stufe erst einmal (das alles ist ein dynamischer Prozeß), dann gibt es verschiedene Wege, die Sie versuchen können, um eine Lösung zu finden. Die hier angegebenen Methoden sind nicht die einzig möglichen. Schrecken Sie also nicht davor zurück, Ihre eigenen anzuwenden. Wichtig ist jedoch, immer einen Notizblock parat liegen zu haben, damit Sie auf ihm neue Einfälle skizzieren können. Das Ziel ist es, das Problem auf eine Reihe von deutlichen und eindeutigen Stufen zur Codierung zurückzuführen (einige Leute neigen dazu, ihr Ziel zu vergessen). Es ist wichtig, das Ziel niemals aus dem Auge zu verlieren. Deshalb sollten Sie die Definition des Problems von Zeit zu Zeit noch einmal lesen.

Als erstes sollte man daran denken, daß es für jedes Problem eine Lösung gibt.

Oftmals haben die härtesten Probleme die elegantesten Lösungen. Es ist wie bei der Mathematik. Oft kommt die Hilfe aus einer Ecke, von der Sie glaubten, daß sie nichts mit der Sache zu tun hat. Betrachten Sie das Problem der Erzeugung eines Bewegungsablaufs auf dem Bildschirm (z. B. das Herabfallen in einem Schacht). Man erreicht dies durch perspektivische Linien, die sich nach außen hin fortbewegen und zwar langsam in der Bildschirmmitte und nach den Seiten hin schneller werdend. Wer hätte gedacht, daß die Lösung in der Anwendung der UDGs (siehe Kapitel 10) liegt. Damit kommen wir zu einem weiteren Punkt – es gibt mehr als eine Lösung. Deshalb lassen Sie sich nicht zu der Denkweise verleiten: „Es gibt nur einen Weg, und der muß etwas mit diesem . . . zu tun haben".

Verfolgen Sie nicht blind Ihr Ziel. Wahrscheinlich müssen Sie zur Lösung des Problems einen indirekten Weg nehmen. Versuchen Sie deshalb nicht verbissen die Lösung in einem Schritt zu erreichen. Zeitschriften sind eine gute Quelle für Anregungen und Probleme. Sie werden überrascht sein, was Sie da alles aufschnappen können, wenn Sie die aufgelisteten Programme und deren Anmerkungen durchblättern. Sie zeigen Ihnen, wie andere ihre Probleme lösen.

Normalerweise gibt es eine bequeme Art und Weise der Lösung von Algorithmen. Sie besteht darin, sich jeden Punkt getrennt vorzunehmen und eine Routine dafür zu schreiben. Das ist wenig befriedigend, aber Sie können dadurch eine Art Ersatzlösung erhalten. Es gibt Umstände, für die das eine gute Alternative ist. Nehmen wir z. B. die Aufgabe, trigonometrische Formeln im Maschinencode für eine dreidimensionale, perspektivische Projektion zu erstellen. Man kann Gleitkommaroutinen schreiben und dann Approximationsformeln verwenden. Das ist natürlich sehr ermüdend. Die Antwort darauf ist, daß die Verhältniswerte sowie nur in 5°-Intervallen benötigt werden.

Deshalb können Sie genauso gut eine Winkeltabelle verwenden. Als Ergebnis erhalten Sie eine schnelle Routine, die einfach und schnell zu schreiben ist.

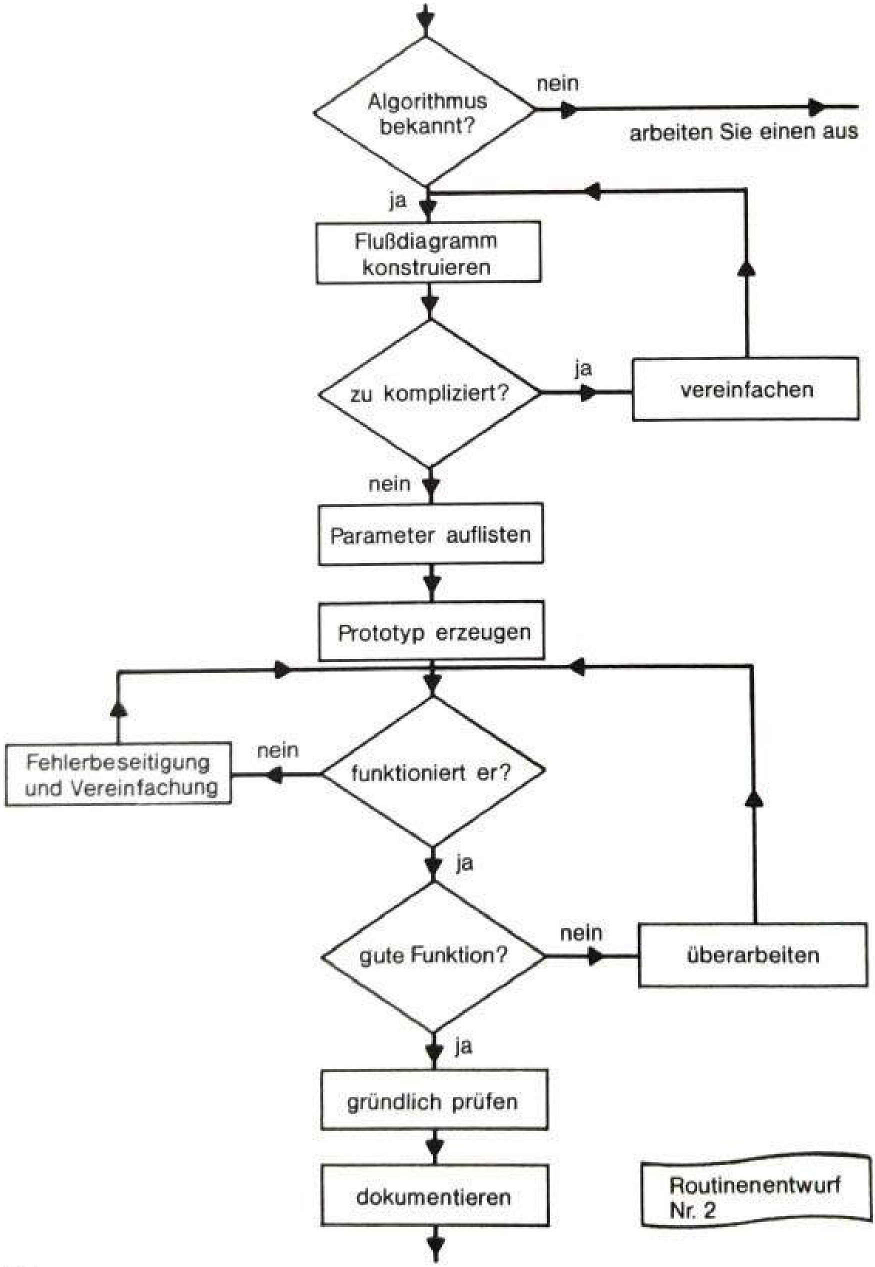
Ich habe eine Schwäche für Ablaufpläne zum Entwerfen von Algorithmen. Sie ermöglichen mir, meine Ideen direkt umzusetzen und ich kann sofort sehen, was passiert. Es ist nichts weiter als ein formales Skizzieren. Dieser Vorgang kann viel Spaß machen, passen Sie daher auf, daß der Ablaufplan nicht zu kompliziert wird. Ich bin der Ansicht, daß Ablaufpläne einen festen Platz im Werkzeugschrank der Programmierer verdienen.

Dieser manuelle Vorgang des Entwurfs eines Algorithmus kann oftmals aufschlußreich sein. Nach einiger Zeit werden Sie feststellen, „daß Sie etwas ohne Nachdenken tun". In dieser Phase müssen Sie weiterforschen, und versuchen, das zugrundeliegende Prinzip herauszufinden. Denken Sie einmal über das Problem der 8 Königinnen nach:

Setzen Sie 8 Königinnen so auf ein Schachbrett, daß sie sich gegenseitig nicht bedrohen.

Schreiben Sie einen Algorithmus, der für die Verwendung im Computer geeignet ist und der das Problem löst. Die Aufgabe erscheint am Anfang etwas zu schwer,

**Abbildung 6.4: Routinenentwurf Nr. 2**



aber wenn Sie versuchen, die Lösung manuell herauszufinden, werden Sie schon nach wenigen Minuten Licht ins Dunkel bringen.
Eine meiner bevorzugten Arten, Probleme zu lösen ist, das Approximationsverfahren. Ich beginne mit einer Idee und verbessere und verfeinere sie nach und nach, bis ich die Antwort finde. Die erste Approximation ist eine Art intelligenten Ratens der Lösung und der Faktoren, die dabei helfen können, das Problem zu lösen. Das Ergebnis kann infolge der vielen Änderungen eine absurde Konstruktion sein, aber das spielt wirklich keine Rolle – solange sie „schlüssig" ist, funktioniert sie und Sie können sie später immer noch verbessern. Denken Sie daran, daß es nur der Prototyp ist.
Damit haben wir einige der wesentlichen Methoden des Algorithmenentwurfs kennengelernt. Verwenden Sie sie freizügig und lassen Sie Ihren Ideen freien Lauf. Die Inspiration kommt manchmal, wenn man sie am wenigsten erwartet. Sie werden wahrscheinlich nicht alle Probleme sofort lösen können, aber es ist auch unwahrscheinlich, daß Sie zu vielen schwierigen Algorithmen gegenüber stehen. Die meisten Probleme, die nicht von den mitgelieferten Service-Routinen unterstützt werden, können schnell zerlegt werden.

# KAPITEL 7

# PRÜFUNG UND FEHLERBESEITIGUNG

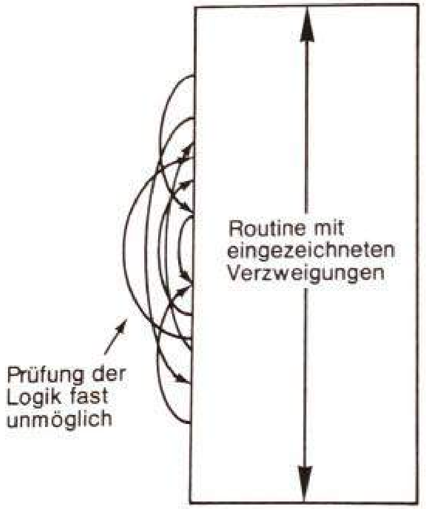
Es gibt kein Maschinencode-Programm ohne seine zugehörenden Fehler. Je länger das Programm ist, desto schlimmer und zahlreicher treten Fehler auf. Man kann nur bei sehr kurzen Routinen erwarten, daß sie auf Anhieb funktionieren. Das vorliegende Kapitel behandelt die Suche und Beseitigung von Fehlern. Es ist entscheidend, daß sie vollständig und endgültig ausgemerzt werden. Der gute Programmierer rechnet mit Fehlern (aber er hofft immer darauf, keine zu machen) und ist daher gut vorbereitet, sie anzugehen.
Fehler können in mannigfaltigen Formen, Größen und Gestalten auftreten. Sie können durch fehlerhafte Annahmen falsche Algorithmen oder durch hundert andere Ursachen verursacht werden. Es gibt nicht so etwas wie einen „Standardfehler", manchmal ist es sogar schwer, die Fehlerursache zu erkennen. Eines der besten Hilfsmittel zur Fehlerbeseitigung ist der Schreibtischtest, ein Testdurchlauf des Programmes mit Papier und Bleistift. Aber vorbeugen ist immer besser als heilen. Aus diesem Grund wurde dieses Kapitel in drei Abschnitte unterteilt. Der erste sagt Ihnen, wie Sie Ihre Routinen einfach halten. Der zweite, wie Sie sie prüfen, und der dritte, wie Sie Fehler beseitigen. Dieses Kapitel ist wahrscheinlich das wertvollste des gesamten Buches. Es bedarf nur weniger Fehler, daß Sie die Lust verlieren und aufgeben. Stellen Sie sich dieser Herausforderung und bekämpfen Sie die Fehler. Schließlich würde das Programmieren im Maschinencode nicht so viel Befriedigung bereiten, wenn es einfach wäre. Die Fehlersuche bei einem Programm kann sogar Spaß machen, wenn Sie dazu in der richtigen Laune sind.

## ROUTINEN EINFACH HALTEN

Eine einfache Routine ist der Traum eines Programmierers. Alles was Sie tun müssen, ist, sie kurz und unkompliziert zu halten und keine schlauen Tricks einzubauen. Es sollten so wenig wie möglich Verzweigungen vorkommen, um die Anzahl der verschiedenen Wege zu minimieren. Zögern Sie nicht, JSR in einer Routine zu verwenden, auch wenn es das einzige Mal ist, daß diese Routine aufgerufen wird. Wird die Routine dadurch einfacher, dann erfüllt JSR seinen Zweck. Im Idealfall sollte eine Routine nicht mehr als 25 Anweisungen enthalten, damit sie alle auf einmal auf den Bildschirm passen. Sobald es mehr werden, sollten Sie anfangen, über den Einsatz von JSRs nachzudenken.
Wenn Sie ein Programm aus dem Kopf schreiben, so wirkt sich das verkrampfend auf Ihre gedanklichen Prozesse aus – sogar in BASIC – da Sie nur 25 Zeilen auf

einmal sehen können. Es ist viel einfacher, mit einem auf Papier aufgeschriebenen Programm zu arbeiten, als mit dem Bildschirm. Sie können das gesamte Programm auf einmal sehen. Sehen Sie sich einige der Routinen in diesem Buch an, dann werden Sie bemerken, daß sie kurz und einfach gehalten sind. Außerdem wurden die Verzweigungen minimiert. Sie werden feststellen, daß sie mit wenigen Ausnahmen ineinander verschachtelt sind. Wenn die Steuerung zwischen den einzelnen Abschnitten einer Routine hin- und herspringt, wie in Abb. 7.1 gezeigt, dann wird es sehr schwierig, die Routine auch nur zu prüfen. Tatsächlich trat kürzlich im Programm einer Zentraleinheit nach zehnjähriger, fehlerfreier Verwendung ein Fehler auf! Halten Sie die Verzweigungslogik einfach, dann wird die Prüfung sehr viel leichter.

**Abbildung 7.1: Wahllose Verzweigung**



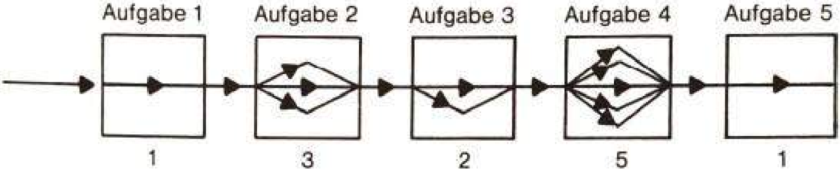

Wahllose Verzweigung

Das Schreiben von unkomplizierten Routinen ist weitestgehend eine Sache der Übung. Wenn Sie sauber planen, werden Sie herausfinden, daß Ihre Routinen automatisch mit einfachen Verzweigungen auskommen. Wenn wir der Baumdiagramm-Theorie folgen, dann werden die Kästen um so komplizierter, je höher sie den Baum hinaufklettern. Wenn schon die Blätter am Boden kompliziert sind, wer kann dann sagen, wie die Spitze aussehen wird?

# PRÜFUNG DER ROUTINEN

Das erste Gesetz der Fehlerbeseitigung ist: „Sie können sich nie sicher sein, daß ein Programm vollständig einwandfrei ist". Das zweite Gesetz lautet: „Einige Fehler sind schwer zu finden und leicht zu korrigieren, einige Fehler sind leicht zu finden, aber schwer zu korrigieren, und einige Fehler gehören in beide Kategorien . . .".
Wenn Sie ein Programm prüfen, dann müssen Sie sicherstellen, daß es für alle im Programm vorkommenden Laufwege funktioniert. Ich meine damit, daß eine bestimmte Routine (z. B. eine Abtastroutine für die Tastatur) verschiedene Wege enthalten kann, die die Maschine befolgen muß. Wird eine Taste gedrückt, dann kann ein Abschnitt des Codes durchgeführt werden, und wird eine andere Taste gedrückt, dann wird vielleicht ein anderer Abschnitt ausgeführt. Daraus ergeben sich zwei verschiedene „Wege", denen das Programm zu folgen hat. Bei der Prüfung dieser Routine müssen wir sicherstellen, daß sie beide Wege richtig durchlaufen kann.

**Abbildung 7.2: Wie viele Steuerungswege?**




Betrachten Sie Abb. 7.2. Sie stellt ein Programm dar, das sich aus 5 verschiedenen Abschnitten zusammensetzt. Nach Beendigung eines Abschnittes geht die Steuerung zur nächsten über. Die Zahl unter jedem Kästchen bezeichnet die Anzahl unterschiedlicher Wege, die die Maschine durch jedes Kästchen nehmen kann. Deshalb ist die Gesamtzahl der unterschiedlichen Wege durch das gesamte Programm das Produkt aus diesen Zahlen, d. h. 1 x 3 x 2 x 5 x 1 = 30. Zur vollständigen Prüfung des Programmes müssen wir jeden einzelnen dieser Wege überprüfen. Das kann ganz schön langweilig werden (denken Sie daran, daß ein vollständiges Programm tausende verschiedener Wege besitzt). Deshalb kann auch ein anderes System zur Prüfung herangezogen werden, das sich „modulare Prüfung" nennt. Es erfordert die Prüfung jedes einzelnen Abschnittes auf allen vorkommenden Wegen. Das heißt, wenn wir wissen, daß der Abschnitt „Spielen" vollständig korrekt ist (oder erscheint), wir ihn abhaken können. Bei Benutzung dieses Verfahrens reduziert sich die Anzahl der Versuche auf die Summe der verschiedenen Wege, d. h. 1 + 3 + 2 + 5 + 1 = 12.
Beachten Sie, wie wir sogar bei diesem einfachen Beispiel unseren Prüfaufwand stark reduzieren konnten. Die modulare Prüfung Ihres Programmes ist oftmals der

natürliche Weg. Sie werden wahrscheinlich, wenn Sie gerade einen Abschnitt des Codes zu Ende geschrieben haben, nicht alle anderen, zugehörenden Routinen auch mit fertiggestellt haben. Prüfen Sie die Routine, solange Sie sie noch im Kopf haben.

Damit kommen wir zu dem Stadium, in dem wir eine spezielle Routine prüfen wollen, aber nicht wissen, wie wir das machen sollen. Vergewissern Sie sich zuerst, daß Sie eine Kopie auf Band oder Diskette abgespeichert haben. Gehen Sie dann wie folgt vor:

1. Untersuchen Sie Ihre Diagramme für diesen Programmabschnitt und nehmen Sie jede Verzweigung oder Sprunganweisung heraus. Es sind dies die einzigen Punkte, an denen sich die Steuerung in zwei Wege aufteilen kann. Jetzt nehmen wir uns vor, den Programmablauf entlang eines jeden dieser Wege laufen zu lassen, indem wir das Programm so „impfen", daß alle Verzweigungen und Sprunganweisungen ausgeführt werden (nicht alle auf einmal), sondern nacheinander). Das „Impfen" eines Programmes ist sehr einfach: Können Sie ein Programm ohne obskure Umstände nicht dazu bringen, bei BCC zu verzweigen, dann fügen Sie davor eine CLC-Anweisung ein (dieses kann die Verzweigungsausgänge umkehren). Das Programm bemerkt den Unterschied nicht. Dieses Verfahren mag wenig elegant aussehen, aber es funktioniert. Vergessen Sie nicht, die eingeimpften Befehle nach der Prüfung wieder zu entfernen. Diese erste Stufe zeigt uns, daß alle Zeilen der Steuerung funktionsfähig sind.
2. Die nächste Stufe ist unangenehmer für das Programm. Sie hat zum Ziel, die Routine durch Eingabe von falschen Daten, die es während des Programmlaufs empfangen konnte, zu verwirren. Sie dürfen hier ruhig rücksichtslos vorgehen. Oftmals ist sich ein Programmierer eines Fehlers im Geiste wohl bewußt, aber er macht sich vor, er sei nicht vorhanden. Werden Sie mit einer solchen Situation konfrontiert, dann müssen Sie zurückgehen und Ihren Fehler korrigieren, da er Sie sonst später heimsuchen wird.

Wenn Ihre Routine mit fliegenden Fahnen bei der Prüfung durchkommt, dann herzlichen Glückwunsch! Wenn ihr jedoch ein Fehler den Garaus macht, dann müssen Sie ihn beseitigen, bevor Sie weitermachen. Denken Sie daran – der Computer macht keine Fehler. Wenn etwas nicht funktioniert, dann ist es alleine Ihnen anzulasten. Ich habe schon oft Programmierer kennengelernt, die einen Fehler bemerken und sich vormachen, daß es ein Maschinenversagen sei. Oftmals tritt ein Fehler nicht jedes Mal auf, wenn ein Programm läuft. Wenn ein Fehler auftritt und sich danach eine Weile versteckt hält, dann begehen Sie nicht den Denkfehler, daß eine Störung des Computers vorliegt. Die Regel ist, wenn Sie einen Fehler einmal bemerken, dann ist er vorhanden und Sie werden ihn auch durch konstantes Ignorieren nicht wieder los. Wenn Sie es sich erst einmal eingestehen, daß sich ein Fehler in Ihrem Code befindet, dann haben Sie ihn schon halb behoben.

# FEHLERBESEITIGUNG

Sie haben also einen Fehler und wissen so ungefähr, wo er sich befindet. Vielleicht haben Sie sogar schon so eine vage Vorstellung, was seine Ursache sein könnte. Außer aufzugeben, haben Sie drei Möglichkeiten: Sie können den Fehler beseitigen, Sie können den fehlerhaften Abschnitt neu schreiben oder Sie können beschließen, daß der Fehler dem Spiel eine neue Dimension hinzufügt, z.B. „Gehen Sie den weißen Klecksen auf dem Bildschirm aus dem Weg. Es sind Wirbel, die bei einem Zusammentreffen die Zeit anhalten. Dadurch wird das Spiel beenden", d.h. die Maschine stoppt. (Zugegeben, das geht ein bißchen weit, aber Sie wären erstaunt über die Anzahl von Programmen, die durch solche göttliche Eingebungen wie diese beeinflußt werden. Sie haben es wahrscheinlich selbst schon einmal gemacht. Obwohl Sie sich davor hüten sollten, es zu oft anzuwenden. Bei manchen Gelegenheiten machen sich solche Fehler sehr gut.) Um diesem hinterlistigen System einen Hauch von Professionalität zu geben, verwenden wir dafür den Ausdruck: „Wenn es nicht funktioniert, dann dokumentieren Sie es (aber erzählen Sie es niemanden)":

Das bevorzugte Verfahren sollte natürlich die Fehlerkorrektur bleiben. Möglicherweise gelingt es Ihnen jedoch nicht, den Fehler aufzuspüren oder schlimmer noch, Sie haben ihn aufgespürt, aber wissen, daß die Beseitigung nicht gerade leicht wird. In diesem (späteren) Stadium sollten Sie sich damit abfinden, den Code neu zu schreiben.

Die erste Stufe der Fehlerbeseitigung ist die Prüfung, ob das, was Sie in den Computer eingegeben haben, mit dem, was Sie eingeben wollten, übereinstimmt. Ein Fehler, der sich aus einem Tippfehler oder Fehldruck ergibt, ist ein „Kobold" (ein entfernter Cousin des Fehlers). Machen Sie diese Prüfung auf jeden Fall zuerst. Sonst pfuschen Sie nur am Programm herum, nur um später festzustellen, daß ein Schreibfehler vorlag – reine Zeitvergeudung.

Der nächste Schritt bei der Fehlerbeseitigung ist die Frage: „Hängt sich das Programm in einer Schleife auf?" Begibt sich die Routine in eine Schleife, dann wird wahrscheinlich einer der beiden Fälle auftreten. Überprüfen Sie als erstes alle Verzweigungsanweisungen, sowohl die bedingten, als auch die unbedingten, da sich das Programm nur über diese in eine Schleife begeben kann. Oftmals werden Sie herausfinden, daß Sie einen relativen Verzweigungsvektor falsch berechnet haben. Ist der Code überall korrekt, dann ist die nächste wahrscheinliche Fehlerursache, daß keine „Beendigungsbedingung" auftritt. Jeder Abschnitt des Codes will die Routine verlassen, wenn die Arbeit getan ist. Das ist der Punkt, an dem die Beendigungsbedingungen ins Spiel kommen: Sie teilen der Routine mit, daß die Arbeit getan ist. Wenn keine Beendigungsbedingungen auftreten, dann haben Sie wahrscheinlich eine Kleinigkeit in Ihrem Programm übersehen (d.h. der Algorithmus ist wahrscheinlich korrekt). Hier wäre also die beste Lösung ein Testlauf des Programmes mit Papier und Bleistift, dann wird der Fehler nach kurzem Kampf

aufgeben. Sollte sich herausstellen, daß Ihr Code immer noch in einer Schleife hängenbleibt und keine dieser Erklärungen zutrifft, dann überprüfen Sie den Code mit einem feinen Sieb: Der Fehler muß irgendwo darin zurückbleiben. Am besten fahren Sie wahrscheinlich mit dem zuvor erwähnten Testlauf, mit Papier und Bleistift. Es ist nur eine Frage der Beharrlichkeit, und sie ist genau das, was den guten Maschinencode-Programmierer ausmacht.
Bleibt Ihr Programm jedoch nicht in einer Schleife hängen, sondern verweigert einfach nur die Arbeit, dann liegt ein Fehler in Ihrer algorithmischen Argumentation vor. Zur Sicherheit überprüfen Sie den Code nochmal auf „Kobolde" und achten Sie besonders auf die Adressiermodi, die Sie verwenden wollten, und die, die Sie tatsächlich verwendet haben. Haben Sie das gemacht und den Fehler immer noch nicht aufspüren können, dann wird es Zeit, diagnostische Hilfen zu verwenden. Diese ermöglichen es Ihnen, den Programmablauf genau zu verfolgen, wodurch Sie mit etwas Glück den Fehler finden. Es gibt eine Reihe unterschiedlicher Hilfsmittel. Deshalb würde ich Ihnen raten, sich jetzt mit ihnen zu befassen, damit Sie das Hilfsmittel, das Ihnen in Ihrer Not am besten hilft, herausfinden.
Das erste ist ein Verfahren, das häufig bei größeren Computern angewendet wird. Es beruht auf dem Abhören des Geräusches, das der Prozessor macht. (ICL-Maschinen sind zu diesem Zweck mit einem Lautstärkeregler für „Heultöne" ausgerüstet). Dieses Geräusch wird an einer Zentraleinheit aufgespürt und verstärkt, so daß die Techniker die Aktivität der Maschinen abhören können. Am C 64 kann dieses Geräusch über das Fernsehgerät hörbar gemacht werden. Sie brauchen nichts weiter tun, als die Lautstärke hochzuregeln und zuzuhören. Wenn Sie SID verwenden, dann können Sie das „Heulen" natürlich nicht hören. Achten Sie auf das Geräusch bei aufgedrehter Lautstärke. Geben Sie einmal die folgende Zeile in BASIC ein:

```
FOR A=1 TO 3000:NEXT
```

und drücken Sie RETURN. Hören Sie den Unterschied? Dies ist das Geräusch, das die Maschine macht, wenn sie sich in einer Schleife befindet. Natürlich hängt die Tonhöhe, die Sie hören, von der Schnelligkeit der Ausführung ab. So können Sie dem Prozessor zuhören, indem Sie die „Heullautstärke" aufdrehen.
Das zweite Verfahren ist eine Art Markensetzung. Betrifft Ihr Code eine Bildschirmaktivität, dann kann es erforderlich sein, eine Verzögerung in die Hauptschleife einzufügen, damit Sie sehen können, was passiert. Das kann man leicht mit der JSR-Anweisung machen. Fügen Sie einfach JSR (Adresse der Verzögerung) in die Schleife ein und lassen Sie den Code laufen. Einzelheiten über das Schreiben einer Verzögerung sind in dem entsprechenden Abschnitt des Kapitels 9 enthalten.
Die Verlangsamung des Programmablaufes kann jedoch nicht ausreichend sein. Es kann sein, daß Sie den Wert eines Registers oder die Distanz, die ein Zeiger auf

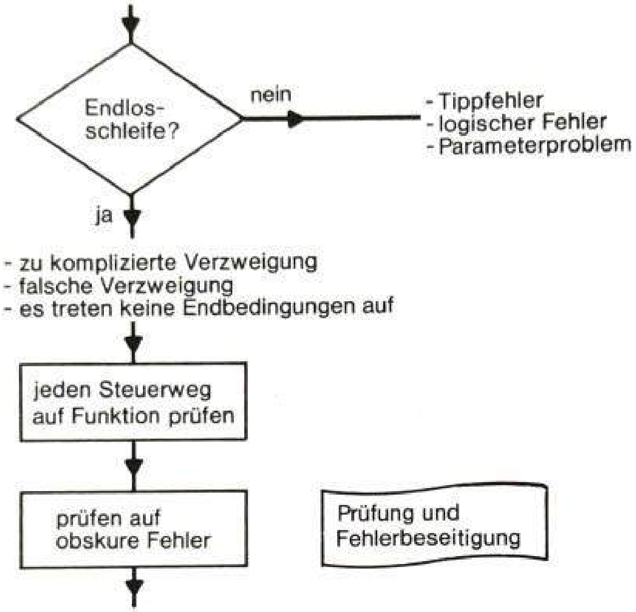
dem Bildschirm zurückgelegt hat, genau sehen wollen. Das erreicht man leicht durch POKEn der Information auf dem Bildschirm vor der Verzögerung und durch ihr anschließendes Löschen. Um zu zeigen, wie weit ein Zeiger auf dem Bildschirm vorangekommen ist, schlage ich vor, seine Adresse mit 160 (weißes Rechteck) zu POKEn und nach der Verzögerung zu entfernen. Beim Programmlauf wandet das weiße Rechteck über den Bildschirm. Wenn Sie den Wert eines Registers wissen wollen, dann verwenden Sie den Modus ABSOLUT X oder ABSOLUT Y, gekoppelt mit der Grundadresse 1024. POKEn Sie wieder ein weißes Rechteck vor der Verzögerung und entfernen Sie es nachher. Aus der Position des weißen Rechtecks kann man auf den Wert des Registers schließen. Befindet sich das Rechteck in der oberen linken Ecke des Bildschirms, dann hat das Register den Wert 0. In der unteren rechten Ecke hat das Register den Wert 39 usw.

Eine viel elegantere Art, Registerwerte festzustellen, ist die Anwendung der vernachlässigten BRK-Anweisung. Sie ist ein von der Software aufgerufener INTERRUPT, der nicht maskierbar ist. Sie können sich also sicher sein, daß er funktioniert. Die Anwendung ist einfach: Sie verstreuen BRKs über Ihr gesamtes Programm, die, wenn sie angetroffen werden, einen INTERRUPT auslösen. Der Vektor ist in 790 und 791 enthalten. Bei älteren CBM-Maschinen zeigt die BRK-Anweisung traditioneller Weise auf den Maschinencode-Monitor. Der C 64 enthält jedoch keinen residenten Monitor, deshalb werden wir nicht viel verlieren. Wir können das Serviceprogramm auf zwei Art und Weisen ablaufen lassen. Wir können entweder die Registerwerte irgendwo sicher abspeichern und dann zu BASIC zurückkehren oder wir können die gedrückten Funktionstasten überprüfen und entsprechend handeln. Wird keine Taste gedrückt, dann läuft das Programm normal ab. F1 verlangsamt das Programm, F2 stoppt es usw. Dieser Punkt wird in dem Abschnitt über den INTERRUPT und der kleine Satz von Hilfsmitteln in Anhang B behandelt. Wir wollen uns zunächst auf die erste Möglichkeit beschränken.

```
:LL0    STA 820
        STX 821
        STY 822
        JSR 58251        (Rückkehr zu BASIC)
```

Legen Sie diese Routine an irgendeiner Stelle ab, z. B. in 49152, und setzen Sie den BRK-Vektor auf die neue Adresse. Nach der Rückkehr zu BASIC können Sie die Registerwerte durch PEEKen von 820 bis 822 herausfinden. Ein BRK ist in Wirklichkeit ein JSR zu einer vorgewählten Adresse. Dabei wird nur ein Programmierbyte benötigt. Ein flexibleres System erstellen Sie sich am besten selbst – siehe Anhang B. Ich hoffe, daß Sie sich Ihr eigenes Hilfsmittel zur Fehlerbeseitigung zurechtschneidern können.
Viel Glück.

**Abbildung 7.3: Prüfung und Fehlerbeseitigung**



# KAPITEL 8

# VERBINDUNGEN: ZUSAMMENFÜGEN DES SPIELS

Nachdem Sie eine Anzahl einzelner Funktionsroutinen fertiggestellt haben, werden Sie feststellen, daß Sie an einem Punkt angelangt sind, wo Sie sie verbinden müssen. Sie können das erkennen, indem Sie das von Ihnen zuvor gezeichnete Baumdiagramm betrachten. Nur die Blätter (Endkästchen) am Baum führen tatsächlich etwas aus. Alle anderen Kästchen stellen nur eine Verbindung zur gewünschten Struktur her. Wenn Sie genug Routinen fertiggestellt haben, dann fangen Sie an, sie zu verbinden, aber erst, nachdem sie gründlich geprüft wurden. Wenn Sie keine Prüfung durchgeführt haben, dann wissen Sie nicht, wo Sie suchen sollen, falls ein Fehler aufgetreten ist. Haben Sie jedoch die einzelnen Routinen auf ihre Funktionstüchtigkeit geprüft, dann können Sie davon ausgehen, daß ein Fehler nur durch ein Parameterproblem oder durch eine fehlerhafte Verbindung verursacht sein kann.

Zuerst betrachten Sie das Kästchen am Baum und finden Sie heraus, welche Verbindungen hergestellt werden müssen. Überprüfen Sie jede Routine und stellen Sie fest, welche Informationen hineingehen und ausgegeben werden. Überprüfen Sie, daß die Routinen nicht miteinander kollidieren. Treten Probleme auf, dann können Sie entweder die Funktionsroutinen abändern (und erneut prüfen) oder den Fehler im Bindeglied flicken. Sie sollten sich aber bereits beim Schreiben der Routinen Gedanken über mögliche Kollisionen machen. Das ist auch der Grund, warum ich immer wieder die Erforschung der exakten Funktion und Beschaffenheit jeder Routine betont habe.

Vergessen Sie nicht, daß weiter oben im Baum ein altes Ergebnis (ein Parameter oder Daten, die vorher erzeugt worden sind) zur Verwendung aufgerufen werden kann. Solange Sie sich dessen bewußt sind, ist alles in Ordnung. Da wir gerade davon reden, den Baum hinaufzuklettern, das Verbindungskästchen muß nicht unbedingt eine einfache Verbindung sein. Es könnte eine Iteration oder eine bedingte Verzweigung oder sogar eine DO-UNTIL-Schleife erforderlich sein. Dieser Abschnitt beschreibt die Stellung dieser Konstruktionen. Natürlich wird zuerst die Einweg-Verbindung behandelt. Sie erfordert einfach, daß alle Routinen nacheinander aufgerufen werden. Die Verbindungsroutine kehrt dann an den Punkt zurück, von wo aus sie aufgerufen wurde.

Jede Routine wird mit einem JSR aufgerufen. Alle Parameter sollten vorher aufgestellt sein und hinterher kann ein bißchen Aufräumen zur Vorbereitung für den nächsten JSR nötig sein. Am Ende der Routine speichern Sie die zukünftig benötigten Daten ab und beenden mit einem RTS. Routinen, die nur einmal zu

verwenden sind, können Ihre Parameter bereits enthalten. Das ist unbedingt vorzuziehen, da es die Lesbarkeit des Programms verbessert. Z.B. ist:

```
JSR KEYCHECK
JSR MOVESHIP
JSR MISSILE
RTS
```

sehr viel deutlicher als

```
LDA #145
STA 121
LDA #34
STA 2
LDX #35
JSR KEYCHECK
LDA 2
AND 121
STA 121
JSR MOVESHIP
LDA 145
EOR #255
STA 2
LDY #40
TYA
JSR MISSILE
STA 2
RTS
```

oder etwa nicht? Wenn Sie sich auf Einfachheit konzentrieren, dann schreiben sich Ihre Programme fast von selbst. Daher ist die Einweg-Verbindung der Kästchen einfach eine Kette von JSRs mit dazwischenliegender Parameteraktivität und einem RTS am Ende.

Ist eine Iteration erforderlich, dann werden die Dinge etwas komplizierter. Fragen Sie sich, was ist die Laufvariable? Ermitteln Sie dann, ob die Anzahl der Iterationen konstant bleibt oder ob diese wiederum ein Parameter ist. Wenn Sie das wissen, fangen Sie mit dem Bindeglied an, indem Sie den Laufwert irgendwo sicher abspeichern. Machen Sie dann so, wie Sie es von der Einmalroutine gewohnt sind, weiter – JSR-Abrufe und Hin- und Herschieben von Parametern. Danach verkleinern Sie die Laufvariable und verwenden BNE, um zum Start des JSR-Aufrufabschnittes zurückzukehren. Da Sie wahrscheinlich weniger als 256 Iterationen

benötigen, ist nur einfache Genauigkeit (ein Byte) erforderlich. Deshalb ist die Konstruktion wie folgt:

```
           AUFSTELLUNG DER LAUFVARIABLEN
:SCHLEIFE  JSR-KETTE UND PARAMETERVERSCHIEBUNG
           VERRINGERUNG DER LAUFVARIABLEN
           BNE-SCHLEIFE
           RTS
```

Das Bindeglied, das eine DO-UNTIL-Konstruktion erforderlich macht, ähnelt dem iterativen Bindeglied sehr. Als erstes müssen Sie herausfinden, was die Betriebsbedingung(en) ist (sind). Alles was Sie tun müssen ist, eine Routine schreiben (wahrscheinlich einen Vergleich), die diese Bedingung prüft, und sie hinter die JSR-Kette setzen. Darauf folgt eine Verzweigungsanweisung, die das Programm aus der Schleife herausbringt, wenn die Bedingung nicht erfüllt worden ist. Das sieht folgendermaßen aus:

```
:SCHLEIFE  JSR-KETTE UND PARAMETERVERSCHIEBUNG
           TESTBEDINGUNG
           FALSCH: VERZWEIGUNG ZUR SCHLEIFE ZURÜCK
           RTS
```

Die Prüfung auf Bedingungen wie '<' und '>' läuft unter bedingter Verzweigung. Das Bindeglied einer bedingten Verzweigung ist wahrscheinlich das komplizierteste Bindeglied dieser Gruppe. Hier können nur die wesentlichen Verzweigungsbedingungen abgedeckt werden. Das sich ergebende Bindeglied kann unübersichtlich werden. Seien Sie also vorsichtig mit Verzweigungen. Es gibt hier zwei Hauptarten von Routinen: Solche, bei denen mehr als eine Routine aufgerufen werden kann, weil zwei Bedingungen erfüllt werden, und solche, bei denen nur eine aufgerufen werden kann. Jede von ihnen kann durch ein Bindeglied weiter aufgeteilt werden, das die gleiche Information bei jeder Bedingung verwendet, und ein anderes, welches unterschiedliche Informationen verwendet. Ich will damit sagen, daß ein Bindeglied abhängig vom Spielstand zu verschiedenen Abschnitten springen kann, während ein anderes den Spielstand, die verbleibende Zeit und andere Faktoren feststellt. Obwohl das so kompliziert erscheint, kann jede Verzweigungsroutine wie folgt geschrieben werden:

```
PRÜFBEDINGUNG 1
BNE
JSR-ROUTINE 1
*
PRÜFBEDINGUNG 2
BNE
JSR-ROUTINE 2
*
usw.
```

BNE wird hier verwendet, um „Verzweigung, falls Bedingung nicht wahr", zu kennzeichnen. Es werden alle Bedingungen nacheinander geprüft und die entsprechende Routine ausgeführt, wenn sie wahr ist. Der Stern '*' zeigt, wo Sie ein JMP (eine erzwungene Verzweigung ist besser) einfügen müssen, falls Sie nur eine Routine laufen lassen wollen. Diese würde die Steuerung am Ende des Bindeglieds ändern. In der folgenden Situation kann dies wünschenswert sein.

Angenommen, Sie haben eine Rangfolgetabelle mit einem von fünf möglichen Kommentaren über Ihren Spielstand. Lassen Sie den '*'-Teil aus, dann müssen Sie für jeden Kommentar prüfen, ob der Spielstand größer und kleiner als zwei andere Zahlen ist, um festzustellen, ob er in einem bestimmten Bereich liegt. Wenn Sie einfach oben anfangen und eine '>'-Überprüfung mit einem '*'-Sprung verwenden, dann müssen sie nur die Bedingung '>' überprüfen.

Werden Programme ausgeführt, die von einem Wert eines abgegrenzten Bereiches (X = 1, 2, 3, 4, 5) abhängen, dann können Sie durch Änderung eines Verzweigungsvektors die Prüfung einer Bedingung umgehen. Nehmen wir an, der Wert im Akkumulator liegt zwischen 0 und 5. Dadurch wird festgelegt, welche Routine ausgeführt wird. Das kann folgendermaßen erreicht werden:

```
ASLA
ASLA
ASLA
STA (Operand der BCC-Anweisung)
CLC
BCC (Operand festgelegt durch STA)
JSR ROUTINE #0
JMP END
NOP
NOP
JSR ROUTINE #1
JMP END
NOP
NOP
JSR ROUTINE #2
JMP END
NOP
NOP
usw.
```

Die Zahl im Akkumulator wird mit 8 multipliziert und das Ergebnis im Operandenbyte des BCC gespeichert. Dadurch springt das Programm zum Anfang der entsprechenden JSR-Anweisung. Die beiden NOPs stellen nur sicher, daß die JSRs 9 Byte voneinander entfernt sind.

Die Überprüfung von Bedingungen wie '<' und '>' erreicht man einfach durch die Verwendung der SBC-Anweisung. Ziehen Sie eine Zahl von der anderen ab und werten Sie die Beziehung aus, indem Sie die Übertragungsmarke oder die negative Marke überprüfen. Die Prüfung auf Gleichheit ist natürlich ein einfacher Vorgang.

Funktioniert das Bindeglied aus irgendeinem Grunde nicht, dann dürfte der Fehler nicht in den Routinen liegen, die es abruft – vorausgesetzt, Sie haben diese geprüft. Entweder ist die Logik im Bindeglied falsch oder die Parameter kollidieren. Es gibt keine Entschuldigung dafür, einen Fehler in dieser Art von Routinen nicht zu finden. Ihre Konstruktion ist so einfach, daß ein Testlauf zum Auffinden der Parameteradressen den Fehler schnell aufspüren sollte. Funktioniert die Routine, dann vergessen Sie nicht, sie zu dokumentieren. Die Dokumentation soll Ihnen später einmal helfen, wenn Sie total vergessen haben, wie Sie das Programm geschrieben haben.

# KAPITEL 9

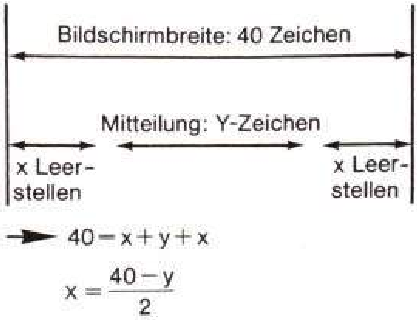
# DIREKTE ROUTINEN

Dieses und das nächste Kapitel enthalten eine große Anzahl von Maschinencoderoutinen, die „allgemeinen Routinen". Jede wird von etwas Text und Vorschlägen für Anpassungsmöglichkeiten begleitet. Es sind mehr als nur Einmalroutinen. Es sind Methoden. Das bedeutet, daß Sie mit ein wenig Nachdenken viele weitere Routinen erzeugen werden. Spielen Sie mit ihnen, mischen Sie sie und experimentieren Sie mit ihnen. Die rechtwinklige Auffüllung (siehe Speicherauffüllung) ist ein gutes Beispiel für die Vielseitigkeit einer anscheinend „normalen" Routine.
Es sind nicht die einzigen Routinen, die in diesem Buch enthalten sind, weitere tauchen an anderen Stellen auf, aber sie sind alle im Routinenindex aufgeführt. Halten Sie in Zeitschriften zwischen den Leserprogrammen Ausschau nach weiteren Routinen dieser Sorte. Sie sind wahrscheinlich an den meisten dieser Programme nicht interessiert, nur an den ein oder zwei ungewöhnlichen Routinen innerhalb dieser Programme.

## TEXT UND LANGSAMER AUSDRUCK

Die meisten Spiele fordern, daß irgendwann Mitteilungen an den Spieler ausgegeben werden. Sie können entscheidend für das Spiel sein oder einfach ein gewisses Maß an Realität beisteuern. Eine einfache Art, Texte auf dem Bildschirm auszugeben, ist die Verwendung der COPY-Routine. Sie kopiert einfach einen Speicherbrocken mit dem jeweils darin enthaltenen Text. Diese Methode ist jedoch sehr primitiv und verschwendet Speicherplatz. Außerdem könnte es sein, daß dieser Text nacheinander in Form einer verzögerten Schreibweise erscheinen soll, wie z. B. die „Game-over"-Mitteilungen am Ende vieler Spielhallenspiele. Vielleicht möchten Sie die Mitteilung auf dem Bildschirm zentrieren, damit das Auge sich darüber freut. Wie erreichen wir das? Wir wollen uns das Problem zuerst in BASIC ansehen.
Nehmen wir an, daß die Mitteilung in M$ enthalten ist. Wir wollen außerdem voraussetzen, daß sich der Cursor bereits in der richtigen Zeile befindet und daß wir nichts weiter tun müssen, als M$ in der Mitte dieser Zeile langsam auszudrucken. Der Bildschirm besitzt eine Breite von insgesamt 40 Zeichen (Abb. 9.1),

**Abbildung 9.1: Textzentrierung**




und die Länge von M$ kann mit LEN (M$) bestimmt werden. Soll die Mitteilung zentriert erscheinen, dann muß vor und hinter ihr die gleiche Anzahl Leerstellen erscheinen. Bezeichnen wir die Anzahl der Leerstellen mit 'X', dann können wir aus dem Diagramm entnehmen, daß:

2x + LEN(M$) = 40

Das Auflösen nach x ergibt

x = (40 − LEN(M$)) ÷ 2

Daher bringt die Anweisung 'PRINT TAB(x);' den Cursor in die druckbereite Position und das ';' hält ihn dort fest. Jetzt müssen wir nur noch die Zeichen nacheinander ausdrucken, wofür wir den Befehl MID$ verwenden und eine Verzögerung nach jedem Zeichen einbauen. Versuchen Sie einmal folgendes:

```
FOR C = 1 TO LEN(M$)
PRINT MID$(MS,C,1);

FOR DE=1 TO 100:NEXT DE
NEXT C
PRINT
```

Vergessen Sie nicht das abschließende PRINT, da sonst der Cursor hinter dem letzten Zeichen der Mitteilung hängenbleibt.

## LANGSAMES DRUCKEN

Das müssen wir also in BASIC machen. Im Maschinencode ist die Mitteilung einfach eine Kette aus Bytes, die mit einem Byte mit vorbestimmten Wert (z.B. 255) abgeschlossen wird und bei der jedes Byte den ASCII-Code des Zeichens wiedergibt. Das Drucken der Zeichen ist einfach. Sie müssen nur eine Schleife erzeugen, die die Daten nacheinander liest (d.h. die Werte in den Akkumulator setzt), und dann eine der ROM-Routinen aufrufen, die ein einzelnes Zeichen druckt. Um den Cursor vorher in die richtige Position zu bringen, ist diese Routine mit einigen Cursorsteuerungen (z.B. Grundposition, 5x Cursor abwärts, 15x Cursor quer) aufzurufen. Vergewissern Sie sich, daß Sie tatsächlich auf den Bildschirm ausgeben, indem Sie die Routine aufrufen, die alle I/O-Kanäle löscht. Sie könnten sonst plötzlich feststellen, daß Sie Zeichen an den Drucker ausgeben! Der Aufruf einer Verzögerungsroutine in der Hauptschleife verlangsamt den Prozeß so, daß das menschliche Auge den Vorgang verfolgen kann. Denken Sie daran, daß Mitteilungen dieser Art sehr gut geeignet sind, den Rythmus des Spielers zu unterbrechen. Die Dateneingabe in diese Routine ist wie folgt: Adresse der Textdatei, Textdatei selbst und Dauer der Verzögerung (obwohl diese bereits festgelegt worden sein kann). Haben Sie eine Anzahl von unterschiedlichen Mitteilungen, dann kann es einfacher sein, die Routine für das langsame Drucken durch 4 oder 5 Sub-Routinen aufrufen zu lassen, die die jeweiligen Parameter für eine Mitteilung festlegen (diese sollten auch den Cursor in Position bringen). Wollen Sie dann eine Mitteilung ausdrucken, so brauchen Sie nur eine der Sub-Routinen aufzurufen. Die Routine gibt nichts an die Sub-Routine, die sie aufgerufen hat, zurück.

Die Adressen der ROM-Routinen, auf die ich mich bezogen habe, sind:

CHROUT: Sie gibt das von dem Wert im Akkumulator dargestellte ASCII-Zeichen an ein Gerät aus. (In unserem Fall an den Bildschirm.) Die Adresse ist 65490($FFD2).

CLRCHIN: Sie bringt alle die I/O-Kanäle in den Normalzustand zurück, d.h., wenn CHROUT verwendet wird, dann werden die Zeichen zum Bildschirm geschickt. Die Adresse ist 65484($FFCC).

Als Variaton dieses Themas zum Ausdrucken von Zeichen können wir eine Leerstelle zwischen die Zeichen setzen. Dadurch „verlängern" wir den Text und lassen ihn größer erscheinen, als er wirklich ist. Vergleichen Sie folgendes:

GAME OVER          G A M E   O V E R

Das zweite Beispiel ist neuartiger und deshalb ansprechender, aber übertreiben Sie es nicht. Man könnte die Nachricht auch unterstreichen oder als effektvollste Möglichkeit von innen nach außen drucken. Diese Methode nennt sich „Metadrucken" und kann sehr effektvoll sein. Aber wie mit allen besonderen Effekten sollte man sparsam damit umgehen und ihn für besondere Gelegenheiten aufheben.

**Metadrucken (BASIC-Programmauszug)**

```
10 L=LEN(M$):IFL/2=INT(L/2)THEN M$=M$+" ":L=L+1
20 X=INT((40-L)/2)
30 FOR P=(INT(L/2)+1)TO1STEP-1
40 PRINT"[CU]"TAB(X+P-1)MID$(M$,P,((INT(L/2)+1)-P)*2+1
50 NEXT P
```

[CU] bedeutet 'Cursor up'

Dieses Programm nimmt M$ und druckt den Inhalt in die Mitte einer Zeile von innen nach außen.
Die folgende Routine druckt die in ASCII-Form gespeicherte Mitteilung beginnend bei START und endend bei einer Eingabe von 13 aus. Die Maximallänge beträgt 255 Byte:

```
:LL0    LDX #0
:LL1    LDA START,X      Lesen
        JSR CHROUT       Ausgeben
        INX Weitergehen
        *
        CMP #13          Rücksprung
        BNE LL1
        RTS
```

Das Zeichen '*' gibt an, wo Sie eine Verzögerungsroutine aufrufen müssen, wenn Sie einen langsamen Druckeffekt erreichen wollen. Das ist auch die Stelle, an der Leerstellen eingesetzt werden können, um den Text ein wenig auseinanderzuziehen. Tun Sie dies, dann vergewissern Sie sich, daß Sie den Wert im Akkumulator aufrechterhalten.
Wenn Sie möchten, daß Ihre Mitteilungen in bestimmten Positionen erscheinen, dann setzen Sie einige Cursorsteuerungsbefehle vor den Text. Ihre Mitteilung könnte z.B. folgendermaßen aussehen:

[HOM][DOWN][DOWN][DOWN] GAME OVER

Unter Verwendung der folgenden Tabelle zur Codierung der Steuerbefehle erhalten wir 19, 17, 17, 17 als erste 4 Eingaben in der Mitteilungstabelle.

| | | | |
|---|---|---|---|
| Cursor-Grundstellung: | 19 | Bildschirmlöschen: | 147 |
| Cursor nach links: | 157 | Cursor nach rechts: | 29 |
| Cursor nach oben: | 145 | Cursor nach unten: | 17 |

## SPEICHERAUFFÜLLUNG

Eine häufig vorkommende Aufgabe für den Programmierer ist das Füllen eines Speichers mit einem bestimmten Wert. Hierbei macht sich die langsame Geschwindigkeit von BASIC besonders bemerkbar. Durch Löschen des Bildschirms erhalten Sie automatisch „Fülle Bildschirm mit 32", aber BASIC geht nicht weiter als das. Bei dem C 64 taucht ein zweites Bildschirmproblem auf: Wenn Sie den Bildschirm löschen, dann wird der Farbspeicher auf die Hintergrundfarbe zurückgesetzt. Das bedeutet, daß Sie zwei POKE-Befehle benötigen, um irgendetwas auf dem Bildschirm anzeigen zu können (Video- und Farb-RAM). Dieses Problem kann man wie folgt lösen. Bevor Sie den Bildschirm löschen, POKEN Sie 53281 mit einem anderen Wert und ändern Sie so die Hintergrundfarbe. Dann löschen Sie den Bildschirm und POKEn 53281 mit seinem vorherigen Wert. Suchen Sie bei größeren Problemen immer nach einfachen Lösungen wie dieser. (Siehe hierzu auch vorherigen Abschnitt über die Löschung des Bildschirms für Text und langsame Ausgabe.)
Im wesentlichen gibt es zwei Arten zum Füllen des Speichers: die Blockauffüllung, wenn Sie nur die Anfangs- und Endadressen angeben sowie die Rechteckfüllung, bei der Sie die Positionen und Abmessungen eines Rechtecks vorgeben (siehe Abb. 9.2(a)). Zum Auffüllen von kleinen Bereichen (<256 Byte) erfüllt eine einfache Schleife mit X- oder Y-Registern im Absolutregister-Modus diese Aufgabe: LDX#0, LDA#CHAR, : LOOP STA ADDR+X, DEX, BNE LOOP, RTS.

## BLOCKAUFFÜLLUNG

Die folgende Routine führt eine einfache Blockauffüllung der Adressen 251–252 bis 253–254 aus (die Endadresse 253–254 wird nicht aufgefüllt). Es gibt eine elegantere Routine, die den Modus Zero-Page + X verwendet, aber die folgende Routine ist besonders einfach aufgebaut:

```
:LL0    LDY #0
:LL2    TXA
        STA(251),Y      ein Byte auffüllen
        INC 251
        BNE LL1         Weitergehen
        INC 252
:LL1    LDA 251
        CMP 253
        BNE LL2
        LDA 252
        CMP 254
        BNE LL2
        RTS
```

Sollten Sie sich über TXA wundern, es dient dazu, das X-Register mit dem zu pokenden Zeichencode zu füllen.

## RECHTECKAUFFÜLLUNG

Die Rechteckauffüllung ist komplizierter, aber nützlicher. Mit ihr kann man schnell eine Grenze erzeugen: Füllen Sie einen Bereich normal und füllen Sie dann ein Rechteck innerhalb dieses Bereiches in einem anderen Code. Das Zentrum wird

**Abbildung 9.2(a): Blockdetails**

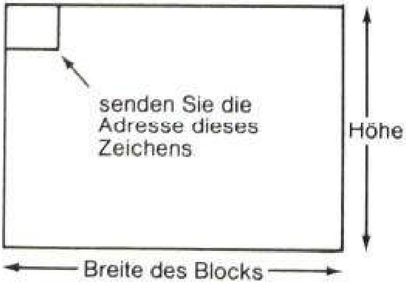
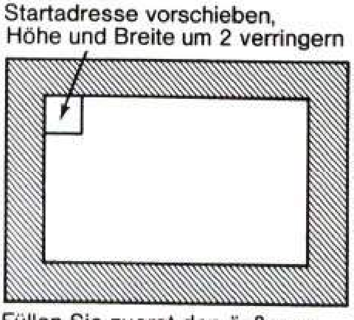
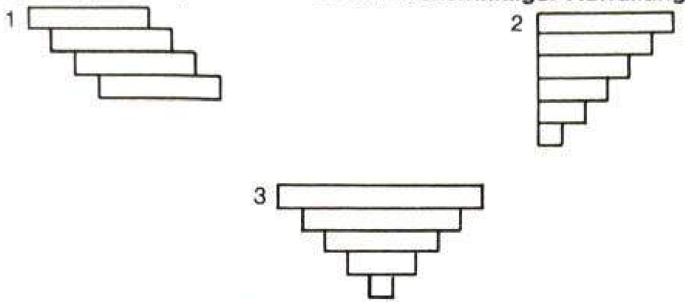



**Abbildung 9.2(b): So erhält man eine Abgrenzung**




Füllen Sie zuerst den äußeren Block. Füllen Sie dann das Innere mit einem anderen Zeichen

**Abbildung 9.2(c): Komplexe Formen mit rechtwinkliger Auffüllung**




1. Tun Sie so, als wäre der Bildschirm 41 Zeichen breit.
2. Verringern Sie jedes Mal um 1.
3. Tun Sie so, als wäre der Bildschirm 41 Zeichen breit und verringern Sie die Breite jedes Mal um 2.

überschrieben und erzeugt so ein Rechteck mit einer Grenze. Wir müssen dazu den zu plottenden Code, die Adresse der oberen linken Ecke, die Breite und die Höhe kennen. Eine weitere Kleinigkeit ist zu betrachten: die Bildschirmbreite, damit sich jede Zeile des Rechtecks direkt über der darüberliegenden befindet (eine Abstufung kann stark ins Auge fallen). Mit dieser Idee im Kopf könnte ein Algorithmus dazu dienen, die Adresse eines jeden Zeilenanfangs zu bestimmen und so viele Zeichen zu plotten, wie die Breite dies vorschreibt. Die folgende Routine tut genau das:

```
:LL1   LDY #0
       LDA CHARCODE
:LL2   STA (251),Y
       INY
       CPY WIDTH            eine Zeile auffüllen
       BNE LL2
       DEC HEIGHT
       BEQ LL3              fertig?
       LDA 251
       CLC
       ADC #SCREEN-WIDTH    nächste Zeile
       STA 251
       LDA 252
       ADC #0
       STA 252
       CLC
       BCC LL1
:LL3   RTS
```

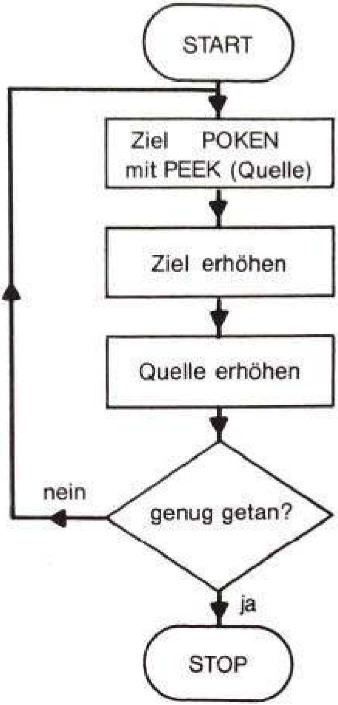
Die Adresse der oberen linken Ecke wird in 251–252 gespeichert. Die HÖHE jedoch wird während des Vorgangs auf Null gesetzt, erwarten Sie deshalb nicht, daß sie ihren Wert behält. Eine dreieckige Auffüllung kann man durch eine stufenweise Verkleinerung (oder Vergrößerung) der Breite erreichen, aber passen Sie auf, daß die BREITE keinen Übergang von 0 auf 255 oder umgekehrt vollführt! Abb. 9.2 zeigt die Flexibilität dieser Routine zur Erzeugung einer Vielzahl von Formen.

## KOPIEREN DES SPEICHERS

BASIC ist beim Zeichnen von Bildern sehr langsam: Sogar das menschliche Auge kann diesem Vorgang ziemlich leicht folgen. Meistens spielt die Geschwindigkeit keine Rolle. Wenn Sie jedoch von einer Anzeige zu einer anderen umschalten oder auch nur den Hintergrund ändern wollen, dann kann einem dadurch das Spiel ziemlich versaut werden. Die Lösung dieses Problems liegt natürlich in der Verwendung des Maschinencodes. Dabei dauert das Kopieren höchstens den Bruchteil einer Sekunde. Die Aufgabe ist sehr einfach zu lösen. Alles, was Sie wissen müssen ist, wo das Kopieren beginnt und endet und wo Sie mit dem Einfügen der kopierten Daten beginnen müssen. Um diesen Vorgang zu erklären, nehmen wir an, daß die Einzelheiten der Titelseite eines Spiels in den Adressen 49152 – 50151 liegen. Um diese auf den Bildschirm zu übertragen, müssen wir nur die Anfangs- und Endpositionen mit 49152 und 50151 angeben und der Routine vorschreiben, die dort gefundenen Daten von der Stelle 1024 an zu kopieren.
Meistens werden Sie diese Routine zum Kopieren von Bildschirmanzeigen oder Abschnitten daraus verwenden. Deshalb bleiben wahrscheinlich alle oder zumindest einige Parameter konstant. Ist dies der Fall, dann zögern Sie nicht, diese in die Routine selbst hineinzuschreiben. Sie müssen darauf achten, daß die beiden Bereiche vollständig voneinander getrennt sind. Wenn irgendeine Speicherstelle sowohl im zu kopierenden, als auch in dem Abschnitt auftritt, in den sie hineinkopiert werden soll, dann könnten Sie Probleme bekommen. Das ist das grundlegende Prinzip des Rollens, das später unter Rollen in diesem Kapitel besprochen wird. Wenn uns diese Routine klar geworden ist, bekommen wir den folgenden Algorithmus:

1) PEEK Adresse von C
2) POKE A, C
3) C = C + 1
4) A = A + 1
5) IF C<E THEN GOTO (1)
6) RETURN

**Abbildung 9.3: Kopieren des Speichers**




In dieser Beschreibung stellen die Buchstaben die folgenden Faktoren dar:

C: ist die erste Adresse, aus der zu kopieren ist. Nach jeder Iteration wird sie erhöht, so daß sie beim nächsten Durchgang die Adresse des nächsten zu kopierenden Zeichens ist.

A: ist die erste Adresse, in die zu kopieren ist. Sie wird genauso wie C erhöht.

E: ist die letzte Adresse, aus der zu kopieren ist.

## Speicherkopie

Das Kopieren von Bildschirmanzeigen ist nicht der einzige Verwendungszweck von COPY-Routinen. Sie können auch dazu verwendet werden, Arrays von Schiffen fremder Eindringlinge in Tabellen als vorbestimmte Angriffswellen aufzustellen oder um vielleicht ein Maschinencode-Programm für sie zu verschieben. Die unten aufgeführte Routine dient zum Kopieren des Bildschirmspeichers, sie könnte aber natürlich auch leicht abgeändert werden. Die Routine gibt keine Parameter aus.

```
:LL0   LDY #0
:LL1   LDA (251),Y    Daten aufnehmen
       STA (253),Y    Kopieren
       SEC
       LDA NOB
       SBC #1         die Anzahl der zu kopierenden Bytes verringern
       STA NOB
       LDA NOB+1
       SBC #0
       STA NOB+1
       INC 251
       BNE LL2
       INC 252
:LL2   INC 253        Weitergehen
       BNE LL3
       INC 254
:LL3   LDA NOB
       CMP #0
       BNE LL1
       LDA NOB+1
       CMP #0         Fertig?
       BNE LL1
       RTS
```

NOB ist die Adresse der zu kopierenden Anzahl von Bytes. Daher ist NOB die Adresse der niederwertigen, und NOB+1 die der höherwertigen Bytes. Das eigentliche Kopieren wird von der zweiten und der dritten Anweisung durchgeführt, während der Rest der Routine den Text zur nächsten Kopierstelle veschiebt. Eine Variante davon könnte das Kopieren eines „Fensters" anstelle eines einfachen Speicherblocks sein. Diese Idee wird in dem Spiel „Laserbike" (siehe Anhang D) verwendet, wo eine 10x10-Zone aus der Spielzone auf den Bildschirm kopiert wird.

Man kann diesen Kopierbefehl vielseitig anwenden. Die Erzeugung eines dauerhaften Hintergrundeffekts wird aber erst durch die Verwendung des Interrupts möglich. Man benötigt dazu nur eine Kopie des Hintergrundes an eine andere Stelle, sowie eine entsprechende Schleifenroutine.

## VERZÖGERUNGEN

Diesen Abschnitt brauchen Sie nicht zu lesen, wenn Sie in BASIC programmieren! BASIC ist, wie ich schon oft gesagt habe, eine sehr langsame Sprache. Da diese Programmiersprache interpretiert (übersetzt) und nicht kompiliert wird, verursacht sie auch überall da Verzögerungen, wo Sie gar keine haben wollen. Wenn Sie eine beträchtliche Verzögerung in BASIC wünschen, dann versuchen Sie die folgende Methode.

```
FOR A = 1 TO 1000: NEXT
```

Dieser Befehl ergibt eine Verzögerung von etwas mehr als einer Sekunde. Was da getan wird, ist eine Zeitverschwendung durch Zählen bis 1000 (in einer Sekunde). Wenn Sie im Maschinencode arbeiten, dann kann es vorkommen, daß Ihr Spiel wirklich zu schnell ist, um damit vernünftig spielen zu können. In dieser Situation benötigen Sie eine Verzögerung. Fügen Sie sie irgendwo in die Hauptschleife ein. Jedesmal, wenn die Schleife ausgeführt wird, verursacht die Verzögerung eine Pause. Natürlich reden wir hier von Verzögerungen in der Größenordnung von Bruchteilen einer Sekunde und nicht von Sekunden. Das grundlegende Prinzip, Zeitverschwendung in einer Schleife, gilt auch hier. Aus irgendeinem unerfreulichen Grund enthält der 6510-Befehlssatz einen Befehl, nichts zu tun! Er ist als „No operation" bekannt und besitzt das Symbol NOP (implizit).
Dieser Befehl kann in einer Zeitverbrauchsschleife benutzt werden, um die Verzögerung zu verdoppeln oder zu verdreifachen. Wir wollen aber zuerst einmal darüber nachdenken, wofür wir diese Verzögerung brauchen.
Der Zweck einer Verzögerung ist die Verlangsamung eines Spiels. Die Beschleunigung eines Spiels während seines Ablaufs ist jedoch (bis zu einem gewissen Ausmaß) ein Mittel, den Spieler unter Druck zu setzen und ihn so zu zwingen, Fehler zu machen. Deshalb wäre es schön, wenn man die Länge der Verzögerungszeit variieren könnte. Wird die Länge der Verzögerungszeit irgendwo in einer 16-Bit-Zahl gespeichert, dann kann die Verzögerungsroutine den erforderlichen Wert PEEKen. Genauso können Sie, wenn Sie die Geschwindigkeit des Spiels beschleunigen wollen, diesen Wert leicht durch Abziehen eines anderen Wertes verringern. Meine Lösung des Problems:

1) Lies die Länge der Verzögerung.
2) Verringere ihren Wert.
3) Führe einige NOPs durch.
4) Ist der Verzögerungszählerstand immer noch >0, dann zurück zu (2).
5) RETURN.

Man kann die Länge der Verzögerungszeit genau voraussagen, indem man die Ausführungszeiten außerhalb der Schleife addiert und dazu die Schleifenzeit, multipliziert mit der Anzahl der Iterationen, hinzuzählt. Ich glaube, daß das beste Verfahren ist, es mit dem Ohr und durch Experimentieren mit verschiedenen Werten auszuprobieren, bis Sie eine zufriedenstellende Lösung finden.
Das folgende Programm führt eine Verzögerung aus, mit deren Länge in der 16-Bit-Zahl der Speicherstelle 253–254 angegeben wird. Dieser Wert bleibt bis zum Ende der Routine unverändert.

```
:LL0    LDA 253
        STA 251
        LDA 254        Aufstellung
        STA 252
:LL1    LDA 251
        SEC
        SBC #1
        STA 251
        LDA 252
        SBC #0
        STA 252
        LDA 251
        CMP #0
        BNE LL1
        LDA 252
        CMP #0
        BNE LL1
        RTS
```

Die folgende Verzögerung, die das X-Register verwendet, ist sehr viel einfacher, aber weniger flexibel:

```
:LL0    LDX #0
:LL1    DEX
        BNE LL1
        RTS
```

Beachten Sie, daß das Register, obwohl es zu Anfang mit Null geladen ist, verkleinert wird, bevor es bei BNE ankommt, wo es in 255 umgewandelt wird. Diese Routine könnte durch Einfügen von ein paar NOPs in der Hauptschleife verlängert werden. Durch Einfügen einer ähnlichen Routine mit dem Y-Register, in diese Routine können wir einen Kompromiß zwischen den beiden Routinen erreichen:

```
:LL0    LDX 254
:LL1    DEY
        BNE LL1
        DEX
        BNE LL1
        RTS
```

Die Länge der Verzögerungszeit wird in Speicherstelle 254 gespeichert.
Sie können Ihre Programme beträchtlich verlangsamen, wenn Sie eine Verzögerungsroutine an den Interrupt anhängen. Denken Sie jedoch daran, daß diese Verzögerung jede sechzigstel Sekunde ausgeführt wird. Machen Sie sie deshalb nicht zu lang. Diese Methode wird mit großer Wirkung zur Fehlerbeseitigung in Programmen verwendet. Eine kräftige Verzögerung verlangsamt BASIC so stark, daß das Löschen des Bildschirms mehrere Sekunden dauern kann. (Aus unerfindlichen Gründen erfolgt das Löschen des Bildschirms von unten nach oben und nicht von oben nach unten.) Setzen Sie diese drei unterschiedlichen Verzögerungsroutinen überall da ein, wo Sie glauben, daß sie nötig sind. Denken Sie daran, daß es im allgemeinen angenehmer ist, eine Verzögerung zu haben, welche die Länge der Verzögerung nicht zerstört (den Parameter, der die Dauer der Pause festlegt).

## AKTUALISIERUNG VON BOMBEN

Sollten Sie einmal eine Version von „Space Invaders" oder dergleichen in BASIC geschrieben haben, dann war Ihr Programm wahrscheinlich ziemlich schnell, bis die Bombardierroutine eingesetzt wurde. Die Bomben verderben immer die Spannung im Ablauf eines jeden BASIC-Spielhallenspieles, da ihre Aktualisierung und Überprüfung so lange braucht. Das ist der Grund, warum einige Leute Spiele ohne Bomben schreiben und stattdessen die Anzahl der Schüsse, die ein Spieler verwenden darf, einschränken. Das ist ein schlechter Ersatz und macht das Spiel langweilig.
Die Antwort darauf ist der Einsatz einer Maschinencoderoutine zur Behandlung der Bombardierung. Sie tut nichts weiter, als den Bildschirm nach Bomben abzusuchen, sie aufzuspüren, zu prüfen, ob sie irgendwo eingeschlagen sind und falls

nicht, sie in ihre nächste Position zu bewegen. Sind sie irgendwo eingeschlagen, so wird das dem BASIC- (oder Maschinencode-) Programm mitgeteilt. Ist dies der Fall, dann muß diese Routine nur noch von BASIC mit einem einfachen SYS aufgerufen und der Inhalt der Marke überprüft werden, um festzustellen, ob die Bombe getroffen hat. Die Bomben können auf zwei Arten bewegt werden. Sie können entweder auf dem Bildschirm gehalten und von dort aus aktualisiert werden, oder sie können in einer Anzeigedatei gespeichert und von dort manövriert werden. Sie müssen entscheiden, welches System Sie verwenden wollen. Am einfachsten ist auf jeden Fall, sie auf dem Bildschirm zu halten. Es hat den Nachteil, daß die Bomben verloren gehen, wenn sie aus Versehen überschrieben werden. Das BASIC-Programm braucht nichts weiter zu tun, als ein Bombenzeichen auf dem Bildschirm in der richtigen Position zu poken. Der Maschinencode nimmt es dann wieder auf und bewegt es weiter. Wenn Sie den Bildschirm jedoch rollen, erhalten die Bomben einen Gleiteffekt, was keine schlechte Sache ist. Verwenden Sie stattdessen das Anzeigedatei-System, müssen Sie sich mit dem Problem herumschlagen, ein Element aus einem Array abzuziehen – eine ganz schöne Murkserei. Einzelheiten über das Anzeigedatei-System sind im Abschnitt über das Bewegen von Objektgruppen (Flottenbewegungen in Kapitel 10) enthalten.
Der Einfachheit halber nehmen wir an, daß Ihre Bomben senkrecht fallen (d. h. von oben nach unten). Dabei braucht das Programm den Bildschirm nur nach dem entsprechenden PEEK-Code abzutasten, die Bombe an der gegenwärtigen Position zu löschen, 40 hinzuzählen und sie dann an der neuen Position wieder anzuzeigen, wobei überprüft wird, ob sie getroffen hat.
Wenn wir darüber nachdenken, werden wir feststellen, daß die Bombe mehrere Dinge treffen könnte. Sie könnte einen Teil des Feindes treffen (d. h. ihre eigene Seite): Das spielt offensichtlich keine Rolle, und die Bombe muß einfach verschwinden. Oder sie könnte einen Schutzzaun treffen (wie z. B. in Space Invaders). Ist dieses der Fall, dann wird normalerweise der Zaun ausgelöscht und die Bombe entfernt. Die Bombe könnte auch auf der Grundlinie des Bildschirmes auftreffen, wobei sie eindeutig ihr nützliches Leben aushaucht. Deshalb können wir sie auch genauso gut zerstören. Einige Programmierer lassen dann, als eine zusätzliche Verfeinerung, ihre Bomben gerne explodieren. Der wichtigste Fall ist natürlich der, in dem die Bombe das Schiff des Spielers trifft. Die Unterscheidung zwischen den Schiffen des Spielers und anderen auf dem Bildschirm dargestellten Ausrüstungen liegt weitgehend beim Programmierer. In „Space Invaders" besteht eine Technik darin, daß Bomben/Schiffskollisionen nur dann überprüft werden, wenn sich die Bombe in der Nähe der Bildschirmunterkante befindet. Das einzige, was die Bombe auf der Zeile, in der sich das Schiff bewegt, treffen könnte, ist das Schiff selbst. Deshalb muß das Programm nur dann eine Kollisionsprüfung durchführen, wenn die Bombe diese Zeile überquert. Bei verzwickteren Situationen besteht das einfachste Nachweisverfahren darin, die PEEK-Codes zu betrachten und sie mit denen des Schiffes und anderer Gegenstände zu vergleichen.

Mit diesen Ausführungen dürften Sie einen allgemeinen Überblick über die Methoden zur Bewegung von Bomben haben, aus der sich ein oder zwei verfeinerte Techniken ableiten lassen. Als erstes müssen Sie den Bildschirm in der entgegengesetzten Richtung der Bombenbewegung abtasten (diese Idee wird auch beim Rollen verwendet), da Sie andernfalls eine Bombe aufgreifen und weitergehen und dabei die gleiche Bombe kurze Zeit später wieder aufgreifen (und wieder weitergehen) werden. Das führt natürlich dazu, daß sich die Bomben in einer Abtastung direkt bis zur Bildschirmunterkante bewegen. Als zweites dürfen Sie nicht vergessen, die Kollisionsmarke zurückzusetzen. Die Art, wie das geschieht, ist sehr wichtig. Vielleicht wollen Sie, daß sich die Bomben in dem BASIC-Programm dreimal so schnell bewegen wie alles andere. Das kann erreicht werden durch die Anwendung von:

```
SYS 826:SYS 826:SYS 826
```

(Dabei wird 826 als Adresse der Routine angenommen). In diesem Fall ist die Verwendung der Maschinencode-Routine zur Rückstellung der Marken zwecklos. Die Marke könnte verloren gehen, wenn eine Kollision in dem ersten oder zweiten Durchgang auftritt.

In diesem Fall ist es der Abrufende, der den Status der Marke aufrechterhalten muß. Denken Sie aber daran, daß dieses nicht immer der Fall sein muß.

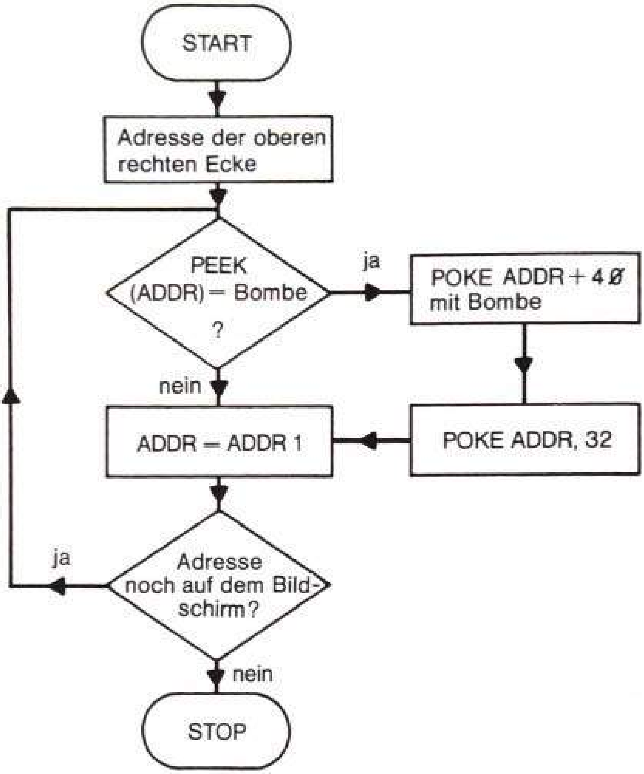
Wir sind jetzt in der Lage, die unterschiedlichen Stadien, die das Verfahren der Bombenaktualisierung durchläuft, festzulegen:

1) **Rückstellen der Kollisionsmarke (kann weggelassen werden).**
2) **Bildschirmabtastung in entgegengesetzter Richtung zur Bombenbewegung.**
3) **Trifft die Bombe etwas, was nicht zur Ausrüstung des Spielers gehört, dann gelten die für diese Situation festgelegten Spielregeln.**
4) **Trifft die Bombe das Schiff des Spielers, dann ist die Kollisionsmarke zu setzen.**
5) **Wiederholung von (2) an, bis der Bildschirm abgetastet ist.**

Möchte der Programmierer eine Explosion auslösen, wenn die Bombe etwas trifft, so kann er dies tun, indem er ein Gatter im SID schaltet.

Zur Richtungsänderung der Bomben müssen Sie nur den Bombensektor ändern. Denken Sie aber daran, daß Sie die Abtastrichtung umkehren müssen, wenn Sie die Bomben in immer tiefere Speicherstellen wandern lassen. Um ein wenig Abwechslung in den Vorgang der Bombardierung zu bringen, versuchen Sie einmal eine „spiralförmige" Bombenroutine, bei der die Bomben ihre Gestalt ändern, während sie fallen. Das macht man am besten durch ändern der UDG-Definition von einer Form in eine andere, jedesmal, wenn die Routine aufgerufen

**Abbildung 9.4: Aktualisierung von Bomben**




wird. Durch den Einsatz dieser Methode habe ich Bomben mit Zähnen erzeugt, die während des Fallvorgangs knirschen. Ehrgeizige Programmierer möchten vielleicht, immer wenn die Bomben sich ändern, noch passende Geräusche hinzufügen. Für Spiele, bei denen die Bewegung der Bomben wirklich schwierig wird, können „zielsuchende" Bomben erzeugt werden. Man erreicht dies, indem man für jede infragekommende Bombe Stücke ihrer Adresse mit denen der „Ziele" vergleicht. Wie man dieses im einzelnen macht, wird unter „Nichtlineare Bewegung" in Kapitel 10 besprochen. Wenn Sie zielsuchende Bomben einsetzen wollen, dann müssen Sie noch ein System erstellen, das zwischen einer aktualisierten Bombe und einer, die noch nicht aktualisiert worden ist, unterscheidet. Eine mögliche Lösung dafür ist, die Bomben bei jeder Abtastung von einem Zeichen in ein anderes und dann wieder zurückzuverwandeln. Auf diese Art und Weise habe ich die Technik der spiralförmigen Bomben entdeckt. Die Bombenroutine benötigt keine Parameter: Sie müssen sie nur aufrufen und die Kollisionsmarke überprüfen.

**Grundlegende Routine zur Bombenaktualisierung**

```
:LL0    LDA #191
        STA 251
        LDA #8          unten anfangen
        STA 252
        LDY #0
:LL1    LDA (251),Y     Zeichen lesen
        CMP #36         Ist es eine Bombe?
        BNE LL2
        LDY #40         eine Zeile weiter nach unten gehen (Vektor +4)
        STA (251),Y     Zurücklegen
        LDY #0
        LDA #32
        STA (251),Y     Anzeigebild zerstören
:LL2    LDA 251
        SEC
        SBC #1
        STA 251         Zurückarbeiten
        LDA 252
        SBC #0
        STA 252
        CMP #3
        BNE LL1         Fertig?
        RTS
```

Diese Routine tastet den gesamten Bildschirm ab und bewegt alle Bomben ($-Zeichen) den Bildschirm herab. Wenn Sie sie in eine andere Richtung bewegen wollen, dann weisen Sie einfach dem Y-Register an den angegebenen Stellen einen anderen Wert zu. Sind Sie an Kollisionen interessiert, dann fügen Sie ein paar Anweisungen für das PEEKen der neuen Stellen ein, bevor die Bombe dorthin bewegt wird, und setzen Sie die Kollisionsmarke entsprechend. Alles, außer den als „$"-Zeichen, bleibt daher unberührt.
Der Interrupt kann gut mit dieser Routine gemeinsam eingesetzt werden. Bilden Sie nur die Bombe auf dem Bildschirm ab, den Rest macht die Maschine für Sie. (Es kann sehr erheiternd sein, jemanden bei dem Versuch zu beobachten, ein „$"-Zeichen auf den Bildschirm zu schreiben, wenn es an den Interrupt angehängt ist. Jedesmal, wenn er es eintippt, fällt es zur Unterkante des Bildschirms.) Das folgende Programm zeigt, wie leistungsfähig diese Routine sogar in BASIC sein kann. POKE 53281,0 dient nur zur Änderung der Hintergrundfarbe (andernfalls

würden sich die Bomben nicht zeigen). Tippen Sie das Programm ab und geben Sie anschließend RUN ein, es lädt den Maschinencode. Geben Sie dann RUN 1000 ein, um den Ablauf der Routine zu verfolgen.

**Stachelhagel**

```
5    REM**STACHELHAGEL
10   DATA 169,191,133,251,169,7,133,252
20   DATA 160,0,177,251,201,36,208,10
30   DATA 160,40,145,251,160,0,169,32
40   DATA 145,251,165,251,56,233,1,133
50   DATA 251,165,252,233,0,133,252,201,3,208,221
     96
60   P = 820
70   READ D:POKE P,D:P=P+1:GOTO 70
1000 PRINT "[SCHIRM NEU]":POKE 53281,0
1010 POKE 1024+INT(40*RND(1),36:SYS 830:GOTO 1010
```

# ROLLEN

Viele der vorhandenen Spiele vermitteln dem Spieler ein Gefühl der Bewegung, indem der Bildschirm in verschiedene Richtungen gerollt wird. Das bedeutet, daß man beim Spielen nicht auf ein Feld von der Größe einer Bildschirmanzeige beschränkt ist. Die Spiele wirken auch viel attraktiver, da mehr geschieht und eine größere Vielseitigkeit vorhanden ist. Eine Rollroutine (SCROLL) in BASIC ist so langsam, daß sie mehrere Sekunden dauert. Es ist deshalb einzusehen, daß Rollroutine im Maschinencode geschrieben werden müssen. Dabei bildet das automatische Rollen, das von der PRINT-Anweisung ausgeführt wird, sobald die Unterkante des Bildschirms erreicht ist, die einzige Ausnahme. Die Anwendung dieser Einrichtung wird im Programm „Asteroiden" demonstriert. Das Rollen kann leicht im Maschinencode aufgerufen werden, indem man entweder die ROM-Routine direkt aufruft, oder indem man ein Zeilenvorschubzeichen zur Druckzeichenroutine schickt.

**Asteroiden**

```
10 REM ASTEROYDS
20 DIM A$(4,4),A(4)
30 A$(1,1)="●"
40 A$(2,2)="╱╲"
50 A$(2,1)="╲╱"
60 A$(3,2)="╱‾╲"
70 A$(3,1)="╲_╱"
80 A$(4,3)="╱‾‾╲"
90 A$(4,2)="|   |"
100 A$(4,1)="╲__╱"
110 A(1)=1:A(2)=2:A(3)=2:A(4)=3
120 IF AP=0THENGOSUB1000
130 PRINTTAB(P) A$(AN,AP):AP=AP-1:GOTO12
   0
1000 AN=INT(4*RND(1)+1):AP=A(AN)
1010 P=INT(33*RND(1)+1):RETURN
```

Ungeachtet dessen, was ich über die Langsamkeit von BASIC gesagt habe, hilft sie uns in diesem Fall sehr gut, Ihnen die grundlegende Funktionshinweise des Rollens zu zeigen. Zum Rollen nach links müssen Sie nur jedes Zeichen in die Position links von ihm kopieren (siehe Abb. 9.5). Tippen Sie das folgende Programm ein, löschen Sie den Bildschirm und zeichnen Sie ein Bild unter Verwendung der Graphik. Lassen Sie das Programm jetzt laufen und beobachten Sie es. Das Bild wird langsam nach links verschoben. Hier ist das Programm:

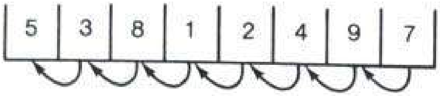
```
10  FOR M=1024 TO 2022
20  POKE M,PEEK(M+1)
30  NEXT M
```

**Abbildung 9.5: Abtastrichtungen**

Nehmen Sie die folgende Zahlenreihe:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 8 | 1 | 2 | 4 | 9 | 7 |

Angenommen, wir wollen sie nach links rollen:




Wir können es nur nicht gleichzeitig tun. Wenn wir es von links nach rechts ausführen, dann funktioniert es:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 8 | 1 | 2 | 4 | 9 | 7 | 7 |

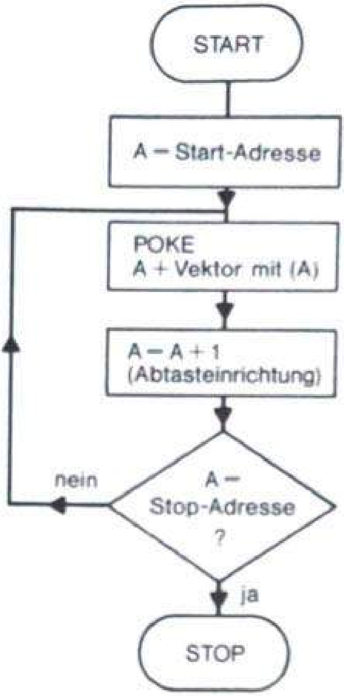
Wenn wir es von rechts nach links ausführen, dann mißlingt es:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

Die „7" wird über den ganzen Weg geschoben. Daraus folgt: Tasten Sie in **entgegengesetzter** Richtung zum Rollen ab.

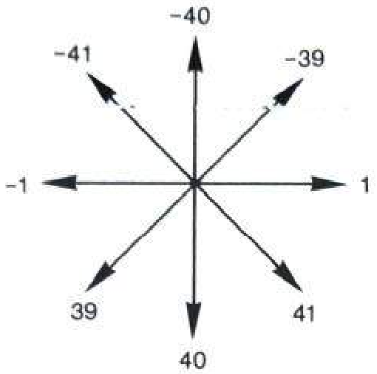
Dieses kleine Programm bringt zwei andere wesentliche Erkenntnisse ans Licht. Erstens: befindet sich etwas ganz links auf dem Bildschirm (z. B. Ihre RUN-Anweisung), dann wird es „herumgewickelt" und bewegt sich auf die äußerste rechte Seite des Bildschirms. Das trifft für jedes Rollen zu, es hängt nur von der Richtung ab. Die andere wichtige Tatsache, die wir aus dem Programm entnehmen können, ist die, daß wir in der dem Rollen entgegengesetzten Richtung „abtasten" müssen. Wir rollten nach links und die FOR-NEXT-Schleife tastete nach rechts ab. Wenn Sie in der gleichen Richtung abtasten und rollen, wird das erste Datenstück im gesamten Bereich wiederholt – probieren Sie es aus. Deshalb arbeitet eine normale COPY-Routine nicht in allen SCROLL-Richtungen.

**Abbildung 9.6: Rollen**




Zusammenfassend kann man sagen, daß das Rollen einfach eine örtliche definierte Kopie ist (d. h. der Bestimmungsort ist nicht weit weg), das Kopieren muß in der entgegengesetzten Richtung zur Bewegung erfolgen und die Bildschirmbegrenzungen müssen evtl. gelöscht werden, um ein Herumwickeln zu vermeiden. Grundsätzlich gibt es 4 Rollrichtungen – nach oben, nach unten, nach links und nach rechts. Diese werden zu den 8 Standardrichtungen kombiniert (Abb. 9.7). Für jede dieser Richtungen können wir eine Zahl angeben, die in relativen Begriffen die Position angibt, in die die Daten zu kopieren sind. Beim Rollen nach rechts ist die Zahl z. B. 1 – jede Speicherstellung wird in ihre Adresse +1 kopiert. Das Rollen nach links besitzt den Wert –1. Ist die Zahl negativ, dann muß der Bereich von links nach rechts abgetastet werden, ist sie positiv, dann muß von rechts nach links abgetastet werden (in fallender Speicherreihenfolge). Diese Zahlen sind alle in dem Diagramm dargestellt (obwohl Sie bereits mit ihnen vertraut sein sollten).

**Abbildung 9.7: Rollvektoren**




Vektor < 0: Abtasten von links nach rechts
Vektor > 0: Abtasten von rechts nach links

Fassen wir alle diese Ideen zusammen, dann ergibt sich der Algorithmus ganz einfach aus der folgenden Methode:

1) M=PEEK (L)
2) POKE L+V,M
3) L=L+D
4) nicht fertig, dann (1)
5) RETURN

Dabei bedeuten die Buchstaben:
M: vorläufiges Kopierregister
L: gegenwärtige Abtastposition
D: Abtastrichtung
V: Rollwert

Um alle Richtungen abzudecken, müssen Sie zwei Routinen haben – die in entgegengesetzte Richtungen abtasten. Die an die Routinen weiterzugebenden Parameter sind die Start- und Stoppadressen sowie der Rollwert. Widerstehen Sie der Versuchung, den gesamten Bildschirm zu rollen, Sie haben dann keinen festen Punkt mehr, um beispielsweise den Spielstand oder die Zahl der Überlebenden anzuzeigen. Eine Technik des Rollens, die ich besonders liebe, ist der in der folgenden Routine gezeigte „Leierkasten". Sie rollt nur (maximal) 256 Bytes, ist

aber sehr handlich, um Mitteilungen über den Bildschirm zu oder ganz allgemein, um Gegenstände in abgegrenzten Bereichen zu bewegen. Der VIC-II-CHIP kann zufälligerweise stetiges Rollen unterstützen. Das bedeutet, daß er den Bildschirm im Hi-RES-Modus (hohe Auflösung) Bit für Bit rollt und nicht Zeichen für Zeichen. Einzelheiten darüber sind in dem Programmierhandbuch angegeben. Sie brauchen trotzdem eine unterstützende Routine zum Rollen von Zeichen, da die Bit-Rollroutine nur für 8 Bit in jede Richtung arbeitet. Danach müssen Sie den gesamten Speicher über die hier angegebene Rollroutine weiterbewegen. Anschließend können Sie zu der Bit-Rollroutine zurückkehren. Die Handhabung von Graphiken im Hi-RES-Modus ist jedoch sehr kompliziert. Sie sollten sie so lange nicht anwenden, bis Sie einige Erfahrung gesammelt haben.

**Kontinuierliches Rollen von 256 Bytes**

```
:LL0    LDX #0
:LL1    INX
        LDA ADDR,X
        DEX
        STA ADDR,X
        INX
        BNE LL1
        RTS
```

Diese Routine rollt 256 Bytes nach links und beginnt bei ADDR. Sie werden in einer geschlossenen Schleife gerollt, so daß alles immer im Kreis geht.

```
:LL0    LDX #0
:LL1    DEX
        LDA ADDR,X
        INX
        STA ADDR,X
        DEX
        BNE LL1
        RTS
```

Diese Routine führt das gleiche nach rechts aus. Ändern Sie LDX #0 auf einen anderen Wert, dann wird die Anzahl der gerollten Bytes reduziert. Dies beseitigt jedoch den „Endloseffekt". Dieses Rollen ist einfach und schnell – es ist überraschend, was man alles mit den X- und Y-Registern erreichen kann.

## ROLLEN IN NIEDRIGERE SPEICHERSTELLEN

Die folgende Routine führt ein Rollen in niedrigere Speicherstellen durch. Jedes Byte wird also in eine niedrigere Adresse verschoben. Deshalb wird der Bereich in der Gegenrichtung abgetastet – in steigender Reihenfolge der Speicheradressen. Jedes Byte wird in die nächstniedrigere Adresse verschoben. Dadurch ergibt sich eine scheinbare Bewegung nach links, wenn man den Ablauf auf dem Bildschirm betrachtet. Die Symbole „LO-START", „HI-START", „LO-STOP" und „HI-STOP" beziehen sich auf die niederwertigen (LO) und höherwertigen (HI) Bytes der START- und STOP-Adressen.

```
:LL0     LDA #LO-START
         STA 251
         LDA #HI-START    Aufstellung
         STA 252
         LDY #0
:LL1     INY
         LDA (251),Y
         DEY              Rollen
         STA (251),Y
         INC 251
         BNE LL2
         INC 252
:LL2     LDA 251
         CMP #LO-STOP
         BNE LL1
         LDA 252          Fertig?
         CMP #HI-STOP
         BNE LL1
         RTS
```

Der Nachteil dieser Routine besteht darin, daß die Zeichen um die linke Bildschirmkante herumgewickelt werden. Dieses Problem löst man durch Löschen dieser Spalte vor dem Rollen. Wenn Sie dieses tun, dann gibt es nichts zum Herumwickeln. Diese Routine sollte aufgerufen werden, bevor das Rollen erfolgt, da sonst die linke Spalte häßlich erscheint. Die Routine könnte durch Verwendung des X-Registers zum Zählen der gelöschten Zeilen verbessert werden. Dadurch benötigt man keine Vergleichsanweisungen mehr. START und STOP sind die Adressen der Bildschirmecken links oben bzw. links unten.

**Zeilenlöschroutine**

```
:LL0    LDA #LO-START
        STA  251
        LDA #HI-START   Aufstellung
        STA  252
        LDY #0
:LL1    LDA #32
        STA  (251),Y       Auslöschen
        CLC
        LDA  251
        ADC #40            eine Zeile nach unten gehen (+40)
        STA  251
        LDA  252
        ADC #0
        STA  252
        LDA  251
        CMP #LO-STOP
        BNE LL1
        LDA  252
        CMP #HI-STOP
        BNE LL1
        RTS
```

Um mit Hilfe dieser Routinen den gesamten Bildschirm nach links zu rollen, benutzen Sie die folgenden Werte für die Symbole. Werden sie nacheinander abgerufen, dann ist die Gesamtdauer immer noch kleiner als eine zwanzigstel Sekunde.

Zeilenlöschung: START = 1024, STOP = 1984
Rollen : START = 1024, STOP = 2023

Beim Rollen kann der Interrupt mit großer Wirkung eingesetzt werden. Ist die Synchronisation jedoch nicht korrekt, dann ergibt sich ein heilloses Durcheinander. Ich neige dazu, nur kleinere Rollfunktionen an den Interrupt anzuhängen, da diese oft leichter zu handhaben sind. Diese Technik könnte dafür verwendet werden, ein endloses Früchteband (oder Meteoritenschwärme) zu erzeugen, das sich ständig um einen bestimmten Bildschirmteil bewegt. Rollen macht Spaß!

# DIALOG

## TASTATUREINGABE

In BASIC werden die Daten normalerweise über die Anweisung INPUT oder GET eingegeben. Im Maschinencode stehen diese Funktionen nicht zur Verfügung.

**Tabelle 9.1: Tabelle der Tastaturcodes, Speicherplatz 197**

| | | | |
|---|---|---|---|
| Keine Taste gedrückt | | 64 | |
| Die Umschalt/Commodore-Tasten haben keine Wirkung | | 60 | |
| Wagenrücklauf (RETURN) | | 1 | |
| F1 | | 4 | |
| F3 | | 5 | |
| F5 | | 6 | |
| F7 | | 3 | |
| , | (verschiebe nach links) | 47 | Diese Tasten verwende ich zur Verschiebung |
| . | (verschiebe nach rechts) | 44 | |
| Q | (verschiebe nach oben) | 62 | |
| A | (verschiebe nach unten) | 10 | |
| 1 | | 56 | |
| 2 | | 59 | |
| 3 | | 8 | |
| 4 | | 11 | |
| 5 | | 16 | |
| 6 | | 19 | |
| 7 | | 24 | |
| 8 | | 27 | |
| 9 | | 32 | |
| 0 | | 0 | |

Es gibt eine Beziehung zwischen dem ASCII-Code und dem Wert, aber sie ist kompliziert. Es ist viel einfacher, zu wissen, welche Taste was bedeutet.
Die einfachste Art und Weise, einen Dialog zwischen Spieler und Computer zu erzeugen, ist die Verwendung der Tastaturabfrage, die automatisch mit dem Interrupt abläuft. (Vergewissern Sie sich, daß Sie ihn nicht abgeschaltet haben). Jede sechzigstel Sekunde (während eines jeden Interrupts) wird die Tastatur überprüft, ob irgendwelche Tasten seit der letzten Prüfung gedrückt wurden. Wurde eine Taste während dieser Zeitspanne gedrückt, dann wird ihr Wert in der Speicher-

stelle 197 gemäß Tabelle 9.1 gespeichert. Wurde seit der vorherigen Abfrage eine neue Taste gedrückt, dann wird diese in dem Puffer aufgezeichnet, der die letzten 10 gedrückten Tasten speichert.
Dieser Puffer beginnt in Speicherstelle 631 und geht bis 640: Die gegenwärtige Puffergröße (d. h. wie viele Zeichen er umfaßt) kann in Speicherstelle 198 abgefragt werden. Ist der Puffer leer, dann enthält diese Stelle eine Null. Da dieses Buch jedoch den Entwurf von Spielen beschreibt, sollte der Programmierer besser die Speicherstelle 197 PEEKen, da diese ihm Auskunft darüber erteilt, welche Taste gerade gedrückt wird. Das PEEKen des Puffers sagt Ihnen nur, welche Tasten gedrückt wurden – egal, ob das gerade eben oder vor einem Jahr passiert ist. Vergessen Sie nicht, daß dieser Puffer sich im Verlauf eines Spieles sehr schnell auffüllen wird (da er nur 10 Zeichen aufnehmen kann). Wenn Sie zu BASIC zurückkehren, stellen Sie sicher, daß er geleert wird, entweder mit dem folgenden Verfahren:

```
FOR A=1 TO 10:GET A$:NEXT A
```

oder vielleicht etwas eleganter:

```
POKE 198,0
```

Dies setzt die Hinweisadresse des Puffers zurück.

## JOYSTICKS

Joysticks machen die Spiele sicherlich spannender, aber schreiben Sie keine Spiele nur für Joysticks oder nur für die Tastatur – Sie wissen ja nicht, wer das Spiel schließlich spielt. (Paddel und Lichtstifte sind so selten, daß es keinen Sinn hat, sie einzubeziehen. Vergessen Sie sie.)
Die beiden Anschlußstellen für den Joystick befinden sich in 56320 und 56321. Für uns sind nur die niederwertigen 5 Bits interessant. Bit 4 ist mit dem Feuerknopf verbunden. Deshalb verwenden Sie einfach AND #16 und danach BEQ oder BNE, um seinen Zustand zu prüfen. Die Bits 0–3 betreffen die Position des Joysticks. Lesen Sie die Anschlußstelle, AND #15 und suchen Sie deren Wert aus Tabelle 9.2. Wird die Richtung als ein Vektor gespeichert, dann ist es einfach, ihn der Adresse Ihres Schiffes oder was sonst damit gesteuert wird, hinzuzuzählen.
Der Wert 255 zieht 1 ab, wenn kein Übertrag vorhanden ist. Die folgende Routine bewegt Sprite Null über Joysticksignale über den Schirm und sucht den Vektor aus der folgenden Tabelle heraus:

[0,0,0,0,0,0,0,0,0,0,1,1,1,255,1,0,0,0,255,1,255,255,255,0,0,0,0,1,0,255,0,0]

**Tabelle 9.2: Vektortabelle**

| ERHALTENER WERT | RICHTUNG | X-VEKTOR | Y-VEKTOR |
|---|---|---|---|
| 0–4 | keine | . | . |
| 5 | rechts und abwärts | 1 | 1 |
| 6 | rechts und aufwärts | 1 | 255 |
| 7 | | 1 | 0 |
| 8 | keine | . | . |
| 9 | links und abwärts | 255 | 1 |
| 10 | links und aufwärts | 255 | 255 |
| 11 | | 255 | 0 |
| 12 | keine | . | . |
| 13 | abwärts | 0 | 1 |
| 14 | aufwärts | 0 | 255 |
| 15 | keine | . | . |

Diese Routine ergibt zusammen mit der Ansteuerungsroutine für Sprites (siehe nichtlineare Bewegung, Kapitel 10) und ein paar Regeln ein einfaches, aber zufriedenstellendes Spiel. Probieren Sie es aus, Sie werden überrascht sein, wie leicht sie zusammenarbeiten. Die folgende Routine hat keine Verzweigungen und läuft direkt durch.

```
:LLO    LDA  56321         Joystickposition lesen
        AND  #15           Schrott unterdrücken
        ASLA               mit 2 multiplizieren
        TAX
        LDA  TABLE,X       Nachschlagetabelle
        CLC
        ADC  53248
        STA  53248
        LDA  TABLE+1,X     X aktualisieren
        CLC
        ADC  53249
        STA  53249         Y aktualisieren
        RTS
```

Sie können diese Routine so abändern, daß sie mit dem Interrupt läuft und den Cursor über den Bildschirm bewegt, um die Eingabe und Pflege Ihrer Programme zu erleichtern. Sollten Sie feststellen, daß die Bewegung in einzelnen Pixeln für ihren Zweck zu langsam ist, dann ändern Sie in der Tabelle die Einsen in Zweien und 255 in 254 oder rufen Sie einfach die Routine zweimal auf.

## UMKEHRUNGEN UND EXPLOSIONEN

Früher oder später wollen Sie in Ihrem Spiel die Illusion erwecken (es ist alles nur Schein), daß das Schiff des Spielers explodiert oder daß gerade die alles vernichtende Bombe explodiert ist. Bei solchen Gelegenheiten soll der gesamte Bildschirm ein großes Feuerwerk zeigen: Für ein paar Sekunden muß der Bildschirm völlig ausflippen.

Das kann man z. B. dadurch erreichen, daß man den Bildschirm umkehrt, „invertiert". Das bedeutet, daß jedes Pixel untersucht und, wenn es eingeschaltet ist, abgeschaltet wird und umgekehrt. Erfolgt dieser Vorgang schnell genug, dann kann der Effekt faszinierend sein (übertreiben Sie also nicht). Er ist ideal zur Einschränkung der Spielersicht geeignet, wenn er z. B. durch Asteroidenschwärme fliegt. Wenn obendrein die Rand- und Hintergrundfarben ziemlich schnell umgeschaltet werden, erhöht das den Effekt. Die Farbadressen sind 53280 und 53281 (Bildschirm und Rand): Wählen Sie eine Farbzahl von 0 bis 15. Unter der Voraussetzung, daß Sie sich im Zeichenmodus befinden, ist die Bildschirminvertierung einfach, da jedes Zeichen in Negativdarstellung einfach das invertierte Zeichen ist. Probieren Sie es aus. Der Commodore bietet Ihnen weitere Hilfe. Sie brauchen nur 128 zu einer Zahl hinzuzuzählen, um sie von Normal- in Negativdarstellung umzuwandeln. Deshalb können Sie hin und her schalten, indem Sie eine EOR-Verknüpfung mit 128 machen. Um den Bildschirm wieder in den Normalzustand zurückzubringen, müssen Sie nur sicherstellen, daß eine gerade Anzahl von Intervierungen durchgeführt worden ist. Es liegt auf der Hand, daß Probleme auftreten, wenn Sie UDGs oder hochauflösende Graphik verwenden. Diese Probleme können aber überwunden werden. Eine Inversion bei hochauflösender Graphik kann sogar im Maschinencode träge erscheinen. Deshalb verwenden Sie am besten die Bildschirm- und Randfarbe. Vergessen Sie nicht, daß der Bildschirm nur sechzigmal in der Sekunde aktualisiert wird. Versuchen Sie also nicht, schneller zu sein. Wenn Sie den Effekt des „Schneiens" auf ein Minimum reduzieren wollen, warten Sie auf ein Bildwiederholungssignal, bevor Sie invertieren. Der Invertierungsvorgang ist überraschend einfach. Tasten Sie nur den Bildschirm ab und verknüpfen Sie jede Adresse über EOR mit 128. Weitere Möglichkeiten bestehen darin, eine Zeile oder eine Zone auf dem Bildschirm blinken zu lassen, um die Aufmerksamkeit des Spielers auf sie zu lenken.

Um die gegebene Routine dafür umzuwandeln, müssen Sie nur die Start- und Stoppadressen ändern. Ein Algorithmus für dieses Problem könnte so aussehen:

```
1)  I=PEEK (C)
2)  I=I IOR 128
3)  POKE C, I
4)  C=C+1
5)  IF C<=S THEN GOTO (1)
6)  RETURN
```

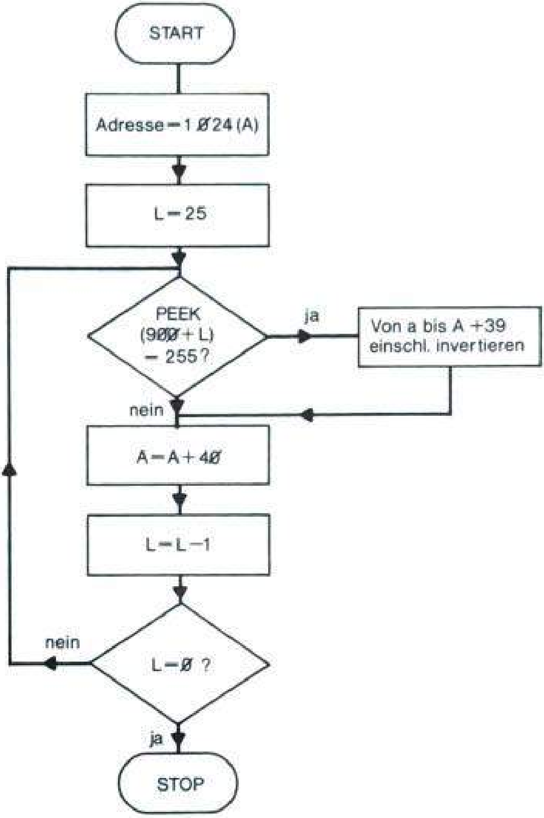
Dabei ist C die Startadresse und S die Stoppadresse. Es werden keine Parameter ausgegeben.

## INVERTIERUNG

Die folgende Routine invertiert einmal den gesamten Bildschirm. Wollen Sie nur einen Abschnitt invertieren, dann ist das durch Änderung der Start- und Stoppadressen möglich. Wenn Sie die EOR-Verknüpfung mit einer anderen Zahl als 128 ausführen, dann kann alles mögliche passieren. Der Bildschirm verwandelt sich plötzlich in ein unsinniges Durcheinander. Die Ordnung wird wieder hergestellt, sobald man die Routine noch einmal aufruft.

```
:LL0    LDA #0        LO-Start-Adresse
        TAX           d.h. LDX #0
        STA 251
        LDA #4        HI-Start
        STA 252
:LL1    LDA (251,X)   Zeichen lesen
        EOR #128
        STA (251,X)   Zurücklegen
        INC 251       Weitergeben
        BNE LL2
        INC 252
:LL2    LDA #7
        CMP #252      HI-Stop-Adresse
        BNE LL1       Fertig?
        LDA #232
        CMP 251       Lo-Stop-Adresse
        BNE LL1       Fertig
        RTS
```

**Abbildung 9.8: Attributsystem**




Hängt man diese Routine an den Interrupt, dann wird es, gelinde gesagt, amüsant. Wenn Sie aber etwas regelmäßig invertieren wollen, dann möchten Sie wirklich nur eine oder zwei Zeilen schreiben. Die folgende Routine macht genau das – Sie invertiert 40 aufeinanderfolgende Zeichen, die bei START beginnen.

```
:LL0    LDX #0
:LL1    LDA START,X     Lesen
        EOR #128        Invertieren
        STA START,X     Zurücklegen
        INX             Weitergeben
        CPX #40         Fertig?
        BNE LL1
        RTS
```

Tatsächlich wäre es leichter, mit X=40 anzufangen, und dann bis 0 hinunterzuzählen. Dadurch fiele der Vergleich weg. Hätten wir jetzt solch eine Routine für jede Bildschirmzeile mit einer Wahlmöglichkeit – blinken oder nicht blinken – und wäre das Ganze an den Interrupt angehängt, dann hätten wir ein Blink-Attributsystem, das dem der Spectrum- und BBC-Maschinen entspricht (Abb. 9.8)! Die folgende Routine tut genau das.

**Attributblinker**

```
:LL0    DEC 254          Interruptzähler um 1 verringern
        BNE LL1          Haben wir bis 60 gezählt?
        LDA #60          Ja, dann blinken und Zähler zurücksetzen
        STA  254
        LDA #255
        STA  251
        LDA #3           oben auf dem Bildschirm anfangen
        STA  252
        LDX #25          alle 25 Zeilen ausführen
:LL3    LDY #40          Jede Zeile besteht aus 40 Zeichen
        LDA TABLE,X
        CMP #255         Diese Zeile blinken lassen?
        BNE LL4
:LL2    LDA (251),Y
        EOR #128
        STA  (251),Y     Ja, also ausführen
        DEY
        BNE LL2
:LL4    CLC
        LDA 251
        ADC #40          Weitergeben
        STA  251
        LDA 252
        ADC #0
        STA  252
        DEX
        BNE LL3          Alles erledigt?
:LL1    RTS              Ja
```

Die Adresse 254 zählt von 60 bis 0 hinunter, wobei sie bei jedem Interrupt 1 abzieht. Daher erfolgt ein Blinken pro Sekunde (tatsächlich ergibt das Zählen von 59 abwärts eine Verzögerung von einer Sekunde).

Erreicht die Adresse 254 die 0, dann wird die Attributtabelle abgetastet und die Zeile mit der Eintragung 255 invertiert. Die Tabelle ist umgekehrt angeordnet. D.h. die erste Eintragung entspricht der 25. Zeile, und Tabelle + 1 entspricht der 24. Zeile usw. Der Interruptanschluß ist nicht enthalten. Er besteht einfach aus einem JMP zum Anfang der IRQ-Serviceroutine.

## EIN ALTERNATIVES SPRITE-SYSTEM

Das residente Sprite-System bewegt die Objekte ziemlich unabhängig vom Bildschirminhalt. Nachfolgend wird eine Routine vorgestellt, die Zeichenblöcke über den Bildschirm bewegt und die für die Bewegung großer Objekte nützlich sein kann. Es gibt keine Einschränkung der Objektgröße oder -anzahl: Die einzige Einschränkung ist, daß nur eine Objektart (d.h. Beschreibung) vorhanden ist.
Die Objekte werden in einer zweidimensionalen Tabelle beschrieben. Die erste Eintragung enthält den POKE-Code jenes Elementes und die zweite enthält den Verbindungsvektor zwischen jenem Zeichen und dem nächsten. Abbildung 9.9 zeigt ein typisches Sinnbild und die entsprechenden Eintragungen. Diese Tabelle wird an den in der Routine gezeigten Adressen gespeichert. Zu ihrer Anwendung POKEn Sie die Adresse in die Zero-Page-Adressen und rufen die Routinen auf. Um das Sinnbild zu löschen, müssen Sie die Löschmarke vor den Aufruf setzen. Diese Marke wird bei Anwendung gelöscht. Eine Objektbewegung erfolgt in folgenden Stufen:

1) Löschen Sie die gegenwärtige Lage.
2) Addieren Sie den Vektor zur Adresse und bewegen Sie so das Objekt.
3) Rufen Sie die Routine erneut auf.

**Abbildung 9.9: Kodieren eines Sprites**

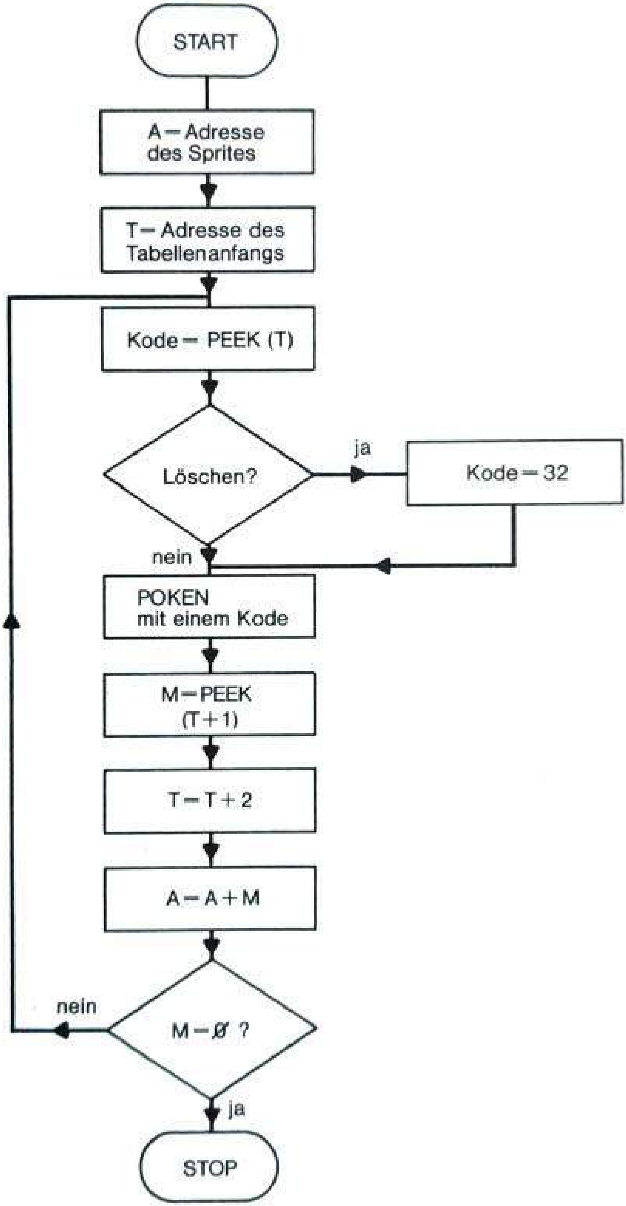
| CODE | VEKTOR | |
|---|---|---|
| 86 | 39 | ' x ' |
| 78 | 1 | ' / ' |
| 37 | 1 | ' % ' |
| 77 | 38 | ' / ' |
| 34 | 1 | ' " ' |
| 34 | 1 | ' " ' |
| 0 | – | 'Null-eingabe ' |

1. Symbol zeichnen
2. Gitter darüber legen
3. Tabelle der Kodes und Vektoren aufstellen
4. Tabellen in der Form: Code, Vektor, Code, Vektor... Null in den Speicher POKEn, z. B. [86,39,78,1,37,1,77,38,34,1,34,1,0]

Wollen Sie damit eine Objektflotte bewegen, dann müssen Sie eine Routine benutzen, welche die zuvor genannten Schritte nacheinander für alle einzelnen Schiffe der Flotte durchführt. Für Einzelheiten über diesen Vorgang siehe den Abschnitt über Flottenbewegungen (Kapitel 10). Diese Routine können Sie für mehrere Arten von Sprites verwenden, wenn Sie die Tabellenadressen zwischen den Aufrufen ändern, aber wahrscheinlich wird das nicht nötig sein (Abb. 9.10). Die folgende Routine ermöglicht die Handhabung von spriteartigen Objekten wie angegeben. TABLE ist die Startadresse der Tabelle. Das Ende der Definition wird mit einem Zeichenwert 0 markiert. Die Definition ist auf max. 127 Eingaben begrenzt (das sollte reichlich genug sein).

```
:LL0    LDA 253
        STA 251
        LDA 254          Adresse einkopieren
        STA 252
        LDX #0           Index der Sinnbildtabelle
        LDY #0
        LDA TABLE,X
:LL2    CPY 2
        BEQ LL3          Sinnbild löschen
        LDA #32          Ja
:LL3    STA (251),Y      Plotten/löschen
        INX
        CLC
        LDA 251
        ADC TABLE,X      Zum nächsten Zeichen übergehen
        STA 251
        LDA 252
        ADC #0
        STA 252
        INX
        LDA TABLE,X
        BNE LL2          Fertig?
        RTS
```

**Abbildung 9.10: Alternatives Spritesystem**



Die Adresse des ersten Spritezeichens wird in die Speicherstellen 253 und 254 übertragen. Diese wird beim Rücksprung nicht geändert. Deshalb kann die Routine wiederholt ohne Aktualisierung aufgerufen werden. Die Option DARSTELLEN/LÖSCHEN erfolgt an der Stelle 2. Ist diese nicht 0, dann wird das Zeichen gelöscht. Beim Rücksprung wird die Stelle 2 immer auf 0 zurückgesetzt. Vergessen Sie nicht, daß man diese Routine an den Interrupt anhängen könnte, um ein bestimmtes Bild auf dem Bildschirm oder, besser noch, ein bestimmtes Vektorsystem aufrechtzuerhalten (siehe Sprites, Kapitel 10). Diese Routine eignet sich gut für das Aufzeichnen schwierig geformter Objekte, wenn eine größere Anzahl davon dargestellt werden muß (siehe Flottenbewegungen, Kapitel 10).

# KAPITEL 10

# INDIREKTE ROUTINEN

## DER INTERRUPT

Was passiert, wenn das Telefon klingelt? Sie unterbrechen, was Sie gerade machen, nehmen den Hörer ab, führen ein Gespräch, und machen dann mit dem weiter, was Sie vorher getan haben. Sie wurden „unterbrochen".
Das ist bei Computern genauso. Sie haben Steuerleitungen, die zur Zentraleinheit führen und eine Unterbrechung (Interrupt) erzeugen können (Abb. 10.1). Wird ein Interrupt aufgerufen, dann speichert der Computer die genauen Einzelheiten darüber, wo er sich gerade befand und was er gerade gemacht hat, im Stapelspeicher. Der Interruptvektor enthält die Adresse einer Routine und die Maschine springt zu dieser Routine. Das Ende der Routine enthält ein RTI – ähnlich wie RTS, bedeutet aber ein Rücksprung von der Unterbrechung (ReTurn from Interrupt). Die Wirkungsweise einer Interruptroutine ist für ein normales Programm vollständig transparent.
Tatsächlich erfolgt ein Interrupt bei BASIC sechzigmal in der Sekunde!

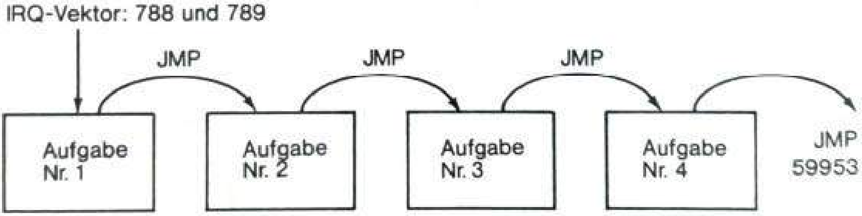
**Abbildung 10.1: CIA Nr. 1 sendet die IRQ-Signale zur Tastenüberprüfung.**

Jede sechzigstel Sekunde erzeugt die IRQ-Leitung (Interrupt ReQuest) einen Interrupt. Die Routine, die ihn ausführt, bringt den Cursor zum Blinken und liest die Tastatur, um festzustellen, welche Tasten Sie drücken. (Nebenbei werden noch andere Aufgaben ausgeführt.) Die Adresse dieser Routinen wird in 788 und 789 gespeichert, wenn Sie es wünschen, können sie auf ihre Routine zeigen lassen. Dabei gibt es ein Problem: Tun Sie dies, dann wird die Tastatur außer Betrieb gesetzt. Sie verlieren den Cursor und die Uhr stoppt – Ihre Verwaltungsroutinen wurden gegen Ihre Routinen ausgetauscht. Sie wirken diesem Effekt entgegen, indem Sie ans Ende Ihrer Routinen nicht RTI, sondern JMP 59953 setzen. Das ist die Adresse des Starts des Interruptserviceprogramms, an dessen Ende sich natürlich ein RTI befindet.

In Wirklichkeit sind drei verschiedene Interrupts verfügbar. Als erstes gibt es den IRQ, den wir bereits besprochen haben, als zweites den NMI, und als drittes den BRK. Der NMI ist dem IRQ sehr ähnlich und wird hier außer acht gelassen. Der BRK ist selbst eine Anweisung und für die Fehlerbeseitigung sehr nützlich (siehe Kapitel 7 Überprüfung und Fehlerbeseitigung). Der Rest dieses Abschnitts widmet sich dem IRQ.

Das hört sich alles ganz gut an, aber Sie könnten fragen, welchen Vorteil ein Interruptsystem gegenüber der normalen Programmierart besitzt. Die Antwort darauf ist, daß es eine Perfektionierung der Zeitsteuerung und Synchronisation ermöglicht. Damit verbunden ist die schöne Eigenschaft, daß etwas an den Interrupt angehängt werden und man es danach vergessen kann. Ich spreche vom „Anhängen", da der Interrupt mehrere, „verkettete" Aufgaben zusammen ausführt (siehe Abb. 10.2). Man kann einfach eine weitere Aufgabe an die Kette „anhängen". Was können wir also alles mit dem Interrupt ausführen? Die Antwort lautet: „So ziemlich alles!"

**Abbildung 10.2: Der Interrupt in Verbindung mit mehreren Routinen**




Sie möchten vielleicht ein bestimmtes Ereignis von Zeit zu Zeit auftreten lassen, wie z.B. das Spielen aufeinander folgender Noten einer Melodie oder die Aktualisierung von feindlichen Schiffen auf dem Bildschirm. In diesem Fall leitet eine einfache Zählroutine nach einer bestimmten Zeit die von Ihnen gewünschte Aktivität ein. Nehmen wir an, Sie wollen z.B. zur Verbesserung Ihres U-Bootspiels alle 4 s einen Impuls („Blip") des Sonars haben. Da der Interrupt sechzigmal in der Sekunde auftritt, müssen Sie zählen: 4x60=240. Wenn wir das Ergebnis der Zählung in Speicherstelle 254 erhalten, kann das Programm folgendermaßen aussehen:

```
        DEC 254
        BNE LL1
        [löst das Gatter bei Sonargeräusch aus]
        LDA #240
        STA 254
LL1     RTS
```

Der Interrupt kann nahezu für alles verwendet werden. Die Anzahl der Interrupts, die pro Sekunde erzeugt wird, kann auch vom Benutzer gesteuert werden. Abschnitt CIA im Anhang C enthält Einzelheiten darüber. Viele der Routinen dieses Buches sind für den Betrieb mit dem Interrupt ausgelegt. Dazu gehören die Vektorbildung für Sprites und das Programmieren der Funktionstasten.
Bei richtiger Anwendung ist der Interrupt der Schlüssel zum Commodore 64. Der VIC-II-Chip besitzt die Fähigkeit, IRQs zu erzeugen: Spritekollisionen sind nur eine von mehreren möglichen Quellen. Sind Interruptroutinen erst mal angehängt, dann kann man sie vergessen. Das macht die Programmierung sehr einfach. Am Anfang wird es Ihnen noch etwas ungewohnt vorkommen, aber Sie werden sich daran gewöhnen und froh sein, daß Sie sich die Mühe gemacht haben.

## ANHÄNGEN

Nachfolgend kurz den Vorgang des Anhängens eines Programms oder einer Routine an den Interrupt beschrieben. Nehmen wir an, daß nur eine Routine angefügt werden soll. Ist dieses nicht der Fall, dann ändern Sie JMP 59953 in die Adresse der nächsten anzuhängenden Routine.

**1) Schreiben Sie die Routine und setzen Sie ein RTS an das Ende.**
**2) Überprüfen Sie sie unter Verwendung einer BASIC-Schleife, die sie dauernd aufruft. In dieser Weise funktioniert der RUN/STOP RESTORE-Trick immer noch.**
**3) Ersetzen Sie RTS durch JMP 59953.**
**4) Richten Sie den IRQ-Vektor mit einer BASIC-Zeile neu aus, z.B.:**
**POKE 788,LO:POKE 789,HI**

Ist alles in Ordnung, dann arbeitet Ihre Routine jetzt sechzigmal in der Sekunde.

## SPRITES

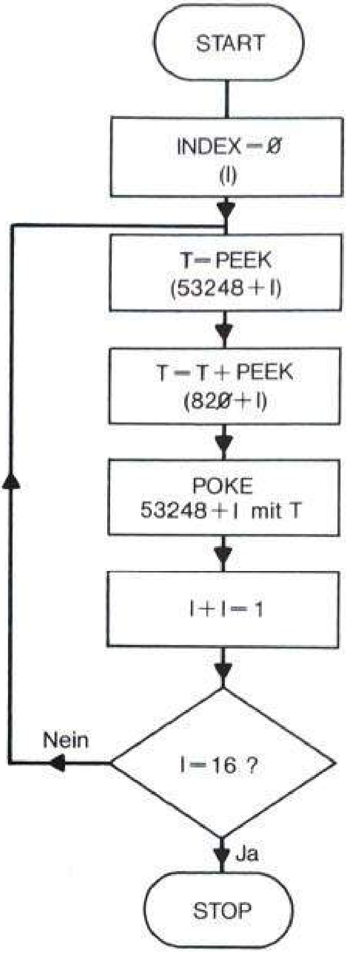
Vor der Ära der Sprites war die zügige Bewegung von Objekten über den Bildschirm eine ermüdende Aufgabe. Glücklicherweise besitzt der Commodore 64 eine kleine Familie von 8 Sprites. Mit sehr wenig Aufwand werden sie für uns agieren, wodurch das Spieleschreiben sehr viel einfacher wird. Sie befinden sich in einer Ebene über oder unter dem Bildschirm, so daß sie sich nicht auf dem Bildschirm selbst bewegen. Sie scheinen über oder unter ihm zu gleiten. Wenn sie sich gegenseitig oder etwas anderes überqueren, dann werden wir durch eine

Eingabe in die Kollisionsprotokolle der Sprites informiert. Durch die Verwendung von Sprites können Sie einen detaillierten Hintergrund zeichnen, ohne sich Sorgen darüber machen zu müssen, daß er verdorben wird.
Die anfängliche Definition von Sprites kann durch die Anwendung einer COPY- oder FILL-Routine beschleunigt werden. Ausführliche Handhabungsdetails für Sprites sind im Handbuch angegeben, und im Anfang dieses Buches in Form einer handlichen Tabelle enthalten. Die „Steuerregister" sind von besonderem Interesse. Es sind Einzelbyte-Register, bei denen jeweils ein Bit einem Sprite zugeordnet ist (Bit 3 = Sprite 3). Diese Register steuern, welche Sprites tatsächlich erscheinen, und welche Sprites ausgedehnt werden und zeichnen u. a. Kollisionen auf. Einzelheiten siehe SID- und VIC-Speicherbelegungspläne in Anhang C. Da die Lage eines Sprites mit Hilfe eines kartesischen Koordinatenpaares (x, y) registriert wird, ist die Zielansteuerung eine sehr einfache Sache (siehe „Nichtlineare Bewegung" weiter hinten in diesem Kapitel). Um eine vollständige Abdeckung des Bildschirmbereichs zu ermöglichen, liegt das höchstwertige Bit der X-Koordinate in einem getrennten Steuerregister. Ich benutze es nicht gerne, weil es die Programme komplizierter macht. Außerdem gibt es, wenn man es nicht verwendet, einen sicheren Bereich auf dem Bildschirm zur Anzeige des Spielstands usw.
Es kommt immer wieder vor, daß ein Sprite mit etwas anderem kollidiert. Passiert dieses, dann wird es in den Kollisionsprotokollen durch Setzen der entsprechenden Bits registriert. Sie werden automatisch zurückgesetzt, nachdem sie gelesen wurden. Deshalb kann es erforderlich werden, diese Daten zu kopieren, wenn Sie sie lesen. Das Protokoll für Kollisionen von Sprites mit Sprites ist 53278 und das für Kollisionen von Sprites mit dem Hintergrund ist 53279. Unter Verwendung dieser Register und des Freigaberegisters können Sie Sprites, die mit etwas zusammengestoßen sind, leicht abschalten. Nehmen wir an, wir wollen alle Kollisionen zwischen Sprites beseitigen. Wir tun das, indem wir das Kollisionsregister PEEKen, alle Einsen in Nullen umwandeln und umgekehrt und dann das Ergebnis mit AND mit dem Freigaberegister verknüpfen:

```
LDA 53278
EOR #255
AND 53269
STA 53269
```

Beide Kollisionsarten werden entfernt, indem man die Kollisionsregister vor der EOR-Anweisung mit OR verknüpft. Daher ist die Aufrechterhaltung von Sprites einfach.
Es bleibt nur noch eines zu tun. Wir müssen ein Sprite in Bewegung setzen und mit eigenem Antrieb in Bewegung halten können. Diese Möglichkeit wäre eine enorme Hilfe in BASIC. Ein schnelles Spielprogramm „Vernichte alles, was sich

**Abbildung 10.3: Durchführung der Spritebewegung**




bewegt", wäre möglich. Die Routine, die dieses kleine Kunstwerk durchführt, gehört zu meinen Lieblingsprogrammen. Ich habe einen ganzen Nachmittag damit verbracht, mit verschiedenen Geschwindigkeiten und Spriteformen herumzuspielen (Abb. 10.3). Mit 16 Bytes ist sie eine der kürzesten Routinen dieses Buches.

```
:LL0    LDX #0
:LL1    LDA TABLE,X
        CLC
        ADC 53248,X
        STA 53248,X
        INX
        CPX #16
        BNE LL1
        RTS
```

Eine Vektortabelle wird in T bis T+15 gespeichert. Für jedes Sprite gibt es einen X- und einen Y-Vektor. Für eine Rückwärtsbewegung müssen Sie nur den Vektor zu 256 hinzuzählen – z. B. wird −1 (nach links) 256+(−1)=255. Für einen wirklich flüssigen Bewegungsablauf sollten Sie diese Routine an den Interrupt anhängen. Dadurch wird das Ganze sechzigmal pro Sekunde aktualisiert. Probieren Sie das folgende Programm einmal an einem Freund aus. Lassen Sie es laufen (RUN) und geben Sie, wenn der Cursor wieder erscheint, den Befehl LIST für das Programm ein – die Sprites wurden zum Leben erweckt!

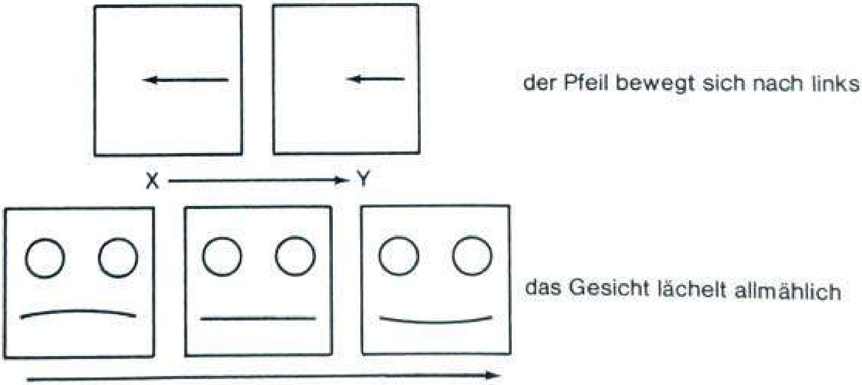
```
10  DATA 162,0,189,52,3,125,0,208,157,0,208,232,224,16,208,242
15  DATA 76,49,234
40  FOR P=49152 TO 49170 : READ D : POKE P,D : NEXT
50  FOR A=2040 TO 2047 : POKE A,15 : NEXT
60  FOR A=960 TO 1022 : POKE A,255 : NEXT
70  FOR A=820 TO 835 : POKE A,INT(4*RND(1)+1) : NEXT
80  POKE 788,0 : POKE 789,192 : POKE 53269,255
```

Wollen Sie alle kollidierten Sprites entfernen, dann setzen Sie die Entfernungsroutine zwischen BNE und RTS ein.

## WEITERE UDG-VERWENDUNGEN

UDGs sind großartig für tolle Graphiken. Ihre Anwendung geht aber weit über die Erzeugung von Fremdlingen hinaus. Betrachten Sie für einen Augenblick die Funktion einer UDG. Sie ermöglicht Ihnen die Erzeugung neuer Zeichen anstelle von alten. Daraus ergeben sich zwei Erkenntnisse: Nichts hält Sie davon ab, ein Zeichenpaar mit der gleichen Definition festzulegen, und nichts hindert Sie daran, dessen Definition zu ändern, nachdem Sie eingegeben worden ist. Verwenden Sie diese beiden Ideen zusammen oder einzeln, dann erreichen Sie sehr leicht, was vorher unmöglich erschien. Wenn Sie die UDGs nicht ganz verstehen, lesen Sie den Abschnitt in Anhang A. Reicht das nicht aus, dann lesen Sie den entsprechenden Abschnitt in dem Programmierhandbuch. Es ist unbedingt erforderlich, daß Sie die Prinzipien, auf denen dieser Abschnitt beruht, völlig verstehen.
Die Idee von zwei identischen graphischen Zeichen ist gar nicht so neu. Schon bevor die UDGs als Attraktion auftauchten, besaßen die PETs zwei identische Zeichen. Dieses waren die Leerstelle und die umgeschaltete Leerstelle. Beide erschienen als normale Leerstelle, ergaben aber die PEEK-Codes 32 und 96. Diese Ähnlichkeit stellt kein Problem für die Maschine dar. Jeder PEEK-Code besitzt eine und nur eine entsprechende Bitmatrix: Das Umgekehrte muß nicht unbedingt stimmen.
Das ist ja alles ganz schön, aber was hat man davon? Eine schnelle Antwort wäre, es ermöglicht die Unterscheidung von Gegenständen, die der Spieler nicht machen kann. Sie könnten so beispielsweise ein paar Fremdlinge zu „Superfremdlingen" machen. Vielleicht sind es die Einzigen, die Bomben abwerfen. Dadurch,

**Abbildung 10.4: UDG-Umschaltung und Matrixübergänge**



daß sie alle gleich aussehen, könnten sie sich unter den Rest der Bande mischen. Unter Anwendung der Technik der umgeschalteten Leerstelle können Sie unsichtbare Barrieren erzeugen. Die Prüfung von deren Überquerung mittels der Adresse ist aufwendig, insbesondere, wenn mehrere Barrieren vorhanden sind. Durch Verwendung unsichtbarer Barrieren wird das Problem gelöst. Diese Idee kann sogar noch weiter ausgebaut werden: Sie können die Bewegung eines Wesens eingrenzen, indem sie es mit einer unsichtbaren Grenze umgeben. Die Technik der gleich aussehenden UDGs ermöglicht Ihnen die Aufstellung von komplexen Bewegungsregeln und Zeichenklassen, ohne dem Spieler allzuviel davon preiszugeben.

Sie sollten jedoch nicht so weit gehen, unsichtbare Minen und andere blöde Dinge zu erzeugen. Der Spieler sollte durch Können und nicht durch Glück überleben. Das Prinzip der Änderung von UDGs ist noch leistungsfähiger, obwohl es ein wenig schwieriger anzuwenden ist. Die erste Anwendung entstammt dem vorausgegangenen Abschnitt. Durch Änderung der Definition eines Gegenstandes, der vorher so aussah wie alle anderen, können Sie diesen plötzlich hervorheben. Diese Technik könnte dazu verwendet werden, dem Spieler einen flüchtigen Blick auf den Gegner zu gestatten oder sogar die Entwicklung des Fortschrittes des Spielers bei der Lösung einer bestimmten Aufgabe anzuzeigen. Während der Ausführung einer bestimmten Tätigkeit zwei Sachen unterscheiden. Man sollte dieses Prinzip jedoch nicht in BASIC versuchen, da es viel zu langsam ist. Der Übergang von einer Form in eine andere erscheint nicht plötzlich, sondern mit einer Art „Kräuseleffekt", der das Spiel verdirbt.

Die fortgeschrittenere Anwendung der UDG-Umschaltung ist der Wechsel der Matrizen, wodurch sich der Eindruck von Bewegung ergibt. Nehmen wir an, das Zeichen X habe das in Abb. 10.4 gezeigte Muster, das dann in Zeichen Y umgewandelt wird. Es hat eine deutliche Linksbewegung stattgefunden. Nehmen wir jetzt an, daß wir eine ganze Zeile oder sogar den gesamten Bildschirm voll von X-Zeichen haben. Durch Änderung der einzelnen Matrix würden Sie gleichzeitig eine Vielzahl von Teilen auf dem Bildschirm geändert haben. Das Ergebnis? Der scheinbare Eindruck einer Bewegung. Sie wirkt weich fließend und ersetzt voll und ganz den Bedarf an Rollroutinen, wenn Sie diesen Effekt nur für den Hintergrund nutzen wollen. Natürlich ist dieses nur die erste und einfachste Anwendung.

Sehen Sie sich die folgende Aufgabe einmal näher an – die Erzeugung des Bildschirms, der eine Jagd durch den dreidimensionalen Raum simuliert. Da hilft Ihnen keine einfache Rollroutine. Sternentrümmer müssen sich in der Bildschirmmitte langsam bewegen, dann nach außen hin beschleunigen und sich am Rand des Bildschirms mit beträchtlicher Geschwindigkeit bewegen. Diese Aufgabe ist mit normalen Methoden nicht leicht zu lösen. Durch Anwendung von UDG-umschaltbaren Methoden und dem Einsatz einer Reihe von Matrixumwandlungen wird die Aufgabe zum Kinderspiel. Man braucht also nicht mit unterschiedlichen Bildschirmbewegungen herumzupfuschen.

Ich habe hier nur eine kurze Einführung in die Welt der UDG-Methoden gegeben. Sie sollte zeigen, daß man die Augen für einfache Lösungen offen halten muß. Die Möglichkeiten, die diese Methoden bieten, sind zahllos, und zweifellos werden viele von Ihnen viele weitere Anwendungen und Tricks für die UDGs entwickeln: Sie sind wundervoll und sollten nicht außen vor gelassen werden, wenn es schwieriger wird. Es macht Spaß, das Gesicht eines Spielers zu beobachten, wenn sich die Fremdlinge bitweise in einen tötlicheren Gegner umwandeln (dieses könnte auch mit BASIC erreicht werden). Außerdem beeinträchtigt es ihre Programme nicht im mindesten, so daß Sie sie vielleicht ein wenig damit würzen möchten. Die gleiche Idee kann auch bei Sprites angewendet werden. Hierbei ist jedoch nur ein POKE nötig, um einen Wechsel in eine vollständig neue Definition durchzuführen. Sie sollten die UDGs nicht nur einfach einmal anwenden und dann für den Rest des Spieles vergessen.

## GERÄUSCHE

Geräusche sind ein wesentliches Element in praktisch jedem verfügbaren Spielhallenspiel. Sie vermitteln dem Spieler eine sehr befriedigende Rückkopplung seiner Aktivitäten. Sie verstärken die Spielerfahrung. Der traditionelle, pochende „Space Invaders"-Rhythmus wurde als Herzklopfen der Invasoren beschrieben. In Wirklichkeit arbeitet das Geräusch gegen den Spieler, da es sich nicht im Takt mit der Bewegung befindet. Es bringt den Spieler aus dem Rhythmus und verhindert so, daß er vollständig mit dem Spiel in Gleichklang gerät.
Der Commodore 64 ist, besonders was Geräusche anbetrifft, begabt. Er besitzt, wie man sagt, den flexibelsten Synthesizer, der gegenwärtig verfügbar ist. Es ist wirklich eine bemerkenswerte Kiste, aber treiben Sie es nicht so weit, daß Sie jeden kleinen Trick, den er ausführen kann, auch einsetzen.
Die Filter haben z. B. keinen brauchbaren Wert für Spiele (oder zumindestens keinen, der bisher ausgenutzt worden ist). Am schönsten ist es, daß Sie eine Geräusch-ADSR-Folge aufstellen können. Sie müssen also nur die Tonsperre lösen, wenn Sie die Reihenfolge noch einmal starten wollen, den Rest können Sie vergessen.
Im wesentlichen gibt es zwei Arten, Geräusche in Spielen einzusetzen: Echtzeit-Geräusche, die während eines Spiels auftreten, um Gewehrschüsse oder Explosionen vorzutäuschen und das nervtötende kleine Liedchen, das immer wieder auf der Titelseite ertönt, um Aufmerksamkeit zu erregen. Wie man Melodien erzeugt, wird später behandelt, jetzt sollen erst mal die Geräusche im Spiel betrachtet werden.

Das Gerät, das die Geräusche im 64 erzeugt, ist als SID (sound interface device) bekannt. Dieser kleine Chip ist selbst ein Computer. Er steuert bis zu drei verschiedene Geräusche gleichzeitig und ermöglicht eine sehr feine Steuerung von Tonhöhe, Lautstärke und Klangfarbe. Wenn Sie den SID wirkungsvoll einsetzen wollen, müssen Sie ihn gut kennen, denn er ist manchmal nicht sehr kooperativ. Die Geräusche kommen aus dem Fernsehgerät oder können über die Monitorleitung in ein HiFi-System eingespeist werden. Der Spieler braucht somit nur das Fernseh-oder HiFi-Gerät leise zu drehen, wenn er die Spielgeräusche abschalten möchte. Das ist sehr nützlich für nächtliche Stunden.
Zur Verwendung des SID müssen Sie eine „Stimme" mit den entsprechenden Werten einrichten und dann die Tonsperre lösen, wenn Sie sie hören wollen. Vergessen Sie nicht, die Lautstärkeregelung (54296) zu POKEn, wobei 15 der Höchstwert ist. Es gibt drei Stimmen, die drei unterschiedliche Geräusche ermöglichen. Möchten Sie mehr haben, dann müssen Sie eine Routine schreiben, welche die notwendigen Werte eingibt und dann das Gatter auslöst. Wenn ich davon spreche, bei einer Stimme ein „Gatter auszulösen", dann meine ich damit, daß man zuerst das Steuerregister mit einer Null POKEt und dann erneut mit dem Wert der Wellenform POKEt. Es reicht nicht aus, nur den Wert der Wellenform zu POKEn. Die Anwendung SID in einem Spiel wie folgt:

1) Initialisieren Sie das SID am Anfang des Spiels.
2) Lösen Sie bei Bedarf die Tonsperre der erforderlichen Stimme.

Es ist keine schlechte Idee, eine Routine zu schreiben, die jede Stimme als eine Subroutine auslöst (auch wenn diese nur wenige Bytes lang ist), da man damit Zeit spart. Tabelle 10.1 zeigt einige mögliche Werte für verschiedene Geräte auf. Eine vollständige Übersicht der Möglichkeiten, die das SID bietet, können Sie dem Programmierhandbuch entnehmen (insbesondere im Abschnitt über die Glockenimitation).

**Tabelle 10.1: Einige Musterhüllkurven**

| ADSR | Wellenform | Klang (Geräusch) |
|---|---|---|
| 0/0/240/0 | 17 | Orgel |
| 5/3/10/0 | 33 | Violine |
| 2/10/0/0 | 17 | Glockenspiel |

| | | | |
|---|---|---|---|
| 5/5/0/0 | 129 | Dampfmaschinen-geräusch | verwenden Sie dafür die Hoch-frequenz 30. |
| 1/10/0/0 | 129 | große Kanone | |
| 5/12/0/0 | 129 | Explosion–hoch=10 | |

Ein Phaser-artiges Geräusch erhalten Sie durch Erzeugung von zwei Geräuschen mit fast der gleichen Frequenz, dadurch erzeugen Sie „Schläge".
Die folgenden Geräusche erhält man durch Verwendung eines Tiefpasses mit einer Resonanz von 15 und einer Stimmenhochfrequenz von 30.

| Filterhochfrequenz | Geräusch |
|---|---|
| 200 | wolkenbruchartiger Regen |
| 140 | großer Waldbrand |
| 110 | Düsenflugzeug |

Das Spielen einer Melodie ist ziemlich einfach, solange sie einstimmig ist. Wollen Sie Akkorde und dergleichen haben, dann wird das Programmieren kompliziert. Am besten verwendet man nur eine Stimme und versucht, wenn man die anderen Stimmen auch erklingen lassen will, diese durch Nachahmen der ersten Stimme in einer anderen Wellenform oder mehrere Oktaven höher zu simulieren. Wie ich bereits angedeutet habe, geht die Verwendung von mehr als einer Stimme, bei der jede ein anderes Stück spielt, über den Umfang dieses Abschnittes hinaus. Dieser Abschnitt sollte nur als befriedigende Übung für den Leser dienen.
Für das Spielen einer Melodie ist dann nur noch eine Tabelle mit Frequenzen und Tondauern erforderlich (sowie eine Nulleingabe für die letzte Note). Das Verfahren ist einfach – lesen Sie eine Note und die Tondauer ein, legen Sie die Frequenz fest und lösen Sie den Ton aus, warten Sie die vorgeschriebene Dauer ab und machen mit der nächsten Note weiter.
Wollen Sie eine Oktave höher spielen, dann verdoppeln Sie nur die Frequenz. Sie können tatsächlich Frequenzen mit jeder Konstanten multiplizieren und erhalten immer noch eine ordentlich klingende Melodie. Versuchen Sie aber nie, eine Konstante zu addieren, da das an einigen Stellen der Melodie zu Halbtonverschiebungen führt. Ein Programm, das eine Melodie spielt, ist nachstehend aufgelistet, und es gibt außerdem ein paar Muster-Melodiedaten. Um die Melodie zu spielen, müssen Sie nur die Routine aufrufen.
Zusammenfassend kann man sagen, daß es leicht ist, Geräusche in Spiele einzubauen. Die Ausarbeitung der „richtigen" Geräusche für ein Spiel ist jedoch subjektiv. Am besten verwendet man ein SID-Auswertungsprogramm (BASIC) und spielt mit etwas Phantasie, bis sich die Geräusche herauszubilden beginnen. Ein Auswertungsprogramm zeigt die charakteristischen Kenngrößen der Stimme an, es ermöglicht Ihnen deren Abänderung und läßt Sie das Geräusch hören. Es ist kurz und einfach, aber sehr nützlich.

**SID-Auswertung**

```
10  REM SID-AUSWERTUNGSPROGRAMM
20  POKE 54296,15:POKE 54273,30
30  VC=[17,33,65,129]
40  INPUT "ADSR";A,D,S,R
50  POKE 54277,D+A*16
60  POKE 54278,R+S*16
70  POKE 54276,0:POKE 54276,VC
80  GOTO 40
```

Dieses Programm fordert Werte für A, D, S und R an und spielt dann das von der Hüllkurve gesteuerte Geräusch. Einfaches Drücken von RETURN (ohne Eingabe) läßt die Hüllkurve unverändert und spielt das Geräusch noch einmal.

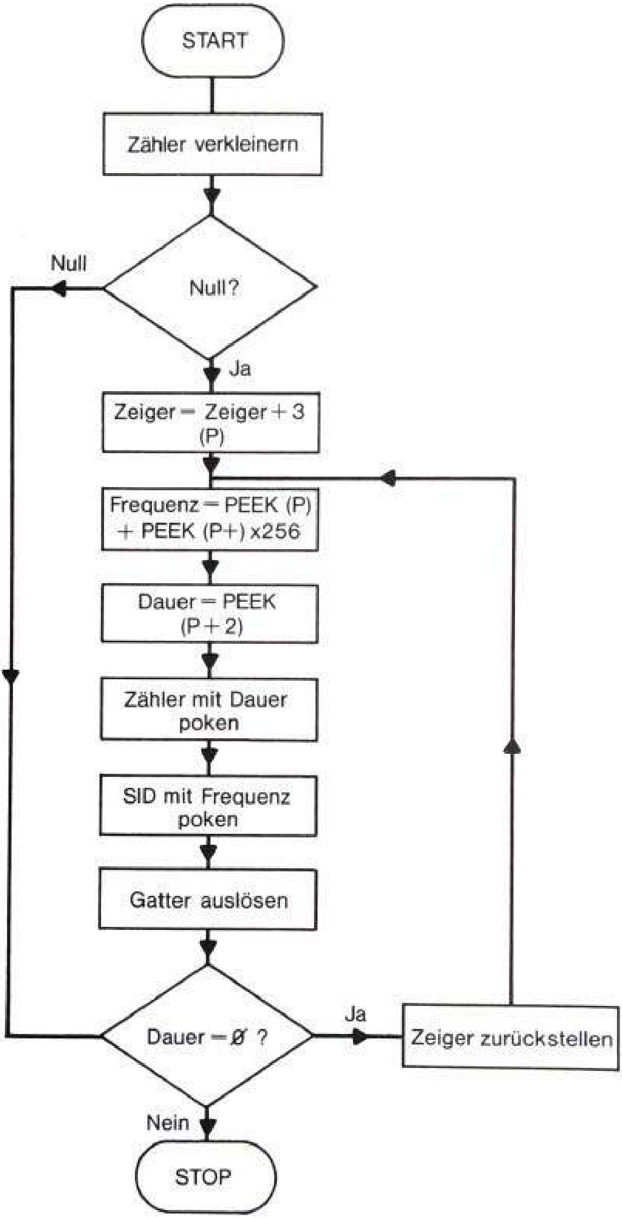
**SID-Schlüsselregister**

| | |
|---|---|
| Anfang von Stimme 1 | 54272 |
| Stimme 2 | 54279 |
| Stimme 3 | 54286 |
| Für jede Stimme: (V = Anfang der Stimme) | |
| V,V+1 | Frequenz |
| V+2, V+3 | Impulswellenformbreite |
| V+4 | Hauptsteuerregister |
| V+5 | Anschlag/Abklingen |
| V+6 | Halten/Ausklingen |
| Lautstärke (0–15) | 54296 |
| Zufallszahlen | 54299 |

Detaillierte Tabelle siehe Anhang C.

Die folgende Routine spielt eine Melodie, die in einer Tabelle gespeichert ist. Jede Note setzt sich aus 3 Bytes zusammen – der tiefen und der hohen Frequenz und der Tondauer. Wenn sie an den Interrupt angehängt ist (wie es sein sollte), dann bedeutet die Dauer 60 eine Note mit einer Länge von 1 Sekunde (Abb. 10.5). Das Ende der Melodie wird durch die Dauer 0 gekennzeichnet. Wird diese erreicht, dann beginnt die Melodie wieder von vorne. Damit das Ende nicht in den Anfang übergeht, setzen Sie eine Verzögerung an das Ende (bestimmbare Dauer, Frequenz 0). Eine solche Melodie an den Interrupt angehängt, kann jemanden zum Wahnsinn treiben (also machen Sie es auch).

**Abbildung 10.5: Interruptbetriebener Melodiespieler**



```
:LL0    DEC 254          Zähler erniedrigen
        BNE LL1          Zeit für eine neue Note?
        CLC              Ja!
        LDA 251
        ADC #3           in der Tabelle weitergehen
        STA 251
        LDA 252
        ADC #0
        STA 252
:LL2    LDY #0
        STY 54276        Geräusch abstellen
        LDA #[Wellenlänge: 17,33,65,129]
        STA 54276        Gatter auslösen
        LDA (251),Y
        STA 54272        niedrige Frequenz einstellen
        INY
        LDA (251),Y
        STA 54273        hohe Frequenz einstellen
        INY
        LDA (251),Y
        STA 254          Dauer einstellen
        BNE LL1          letzte Note?
        LDA #LO-START    Ja, also wiederholen
        STA 251
        LDA #HI-START
        STA 252
        CLC
        BCC LL2
:LL1    RTS
```

HI- und LO-Start bedeutet die Adresse des Melodieanfangs. Wenn Sie „moderne" Musik erzeugen wollen, dann versuchen Sie einmal, das Programm mit vollständig zufälligen Daten laufen zu lassen – Wahnsinn! Für eine anregendere Melodie zeigt Ihnen Tabelle 10.2 ein Melodienmuster.

Diese Daten versuchen, an eine Melodie zu erinnern. Sie sind in der folgenden Reihenfolge für das Spielprogramm zu speichern: niedrige Frequenz, hohe Frequenz, Dauer. Die Dauer wird in Sekunden angegeben, multiplizieren Sie sie also mit 60.

**Tabelle 10.2: Daten für eine Mustermelodie**

| LO | HI | DAUER (mit 60 multiplizieren) |
|---|---|---|
| 96 | 22 | 1 |
| 49 | 28 | 1,5 |
| 19 | 28 | 0,5 |
| 156 | 26 | 1 |
| 96 | 22 | 1 |
| 223 | 29 | 1 |
| 223 | 29 | 1 |
| 49 | 28 | 1 |
| 156 | 26 | 1 |
| 49 | 28 | 1 |
| 135 | 33 | 1 |
| 135 | 33 | 1 |
| 165 | 31 | 1 |
| 135 | 33 | 3 |
| 49 | 28 | 1 |
| 162 | 37 | 1,5 |
| 135 | 33 | 0,5 |
| 223 | 29 | 0,5 |
| 49 | 28 | 1 |
| 156 | 26 | 1 |
| 96 | 22 | 1 |
| 31 | 21 | 1 |
| 49 | 28 | 1 |
| 156 | 26 | 1 |
| 96 | 22 | 1 |
| 96 | 22 | 1 |
| 31 | 22 | 1 |
| 96 | 22 | 3 |
| 0 | 0 | 255 |
| 0 | 0 | 0 |

## GROSS ANGELEGTE SPIELE

Viele Spiele machen auf einem normalgroßen Bildschirm Spaß, aber nach und nach erscheinen sie irgendwie in ihrer physikalischen Größe beschränkt zu sein. Ist dieses der größte verfügbare Spielbereich, dann ist es schwer, das Interesse des Spielers über einen langen Zeitraum aufrechtzuerhalten. Dieses Problem kann jedoch auf mehreren Wegen umgangen werden.

Eine gängige Methode ist die Aufteilung des Spiels in mehrere (oder viele) getrennte Bildschirmbilder. Der Spieler arbeitet sich in einer vorher festgelegten Reihenfolge durch diese Bilder und erhält dadurch den Eindruck, daß er nach und nach durch ein Phantasieland reist. Die andere populäre Methode besteht darin, den Bildschirm wie ein Fenster zu einem sehr viel größeren Spielbereich wirken zu lassen. Dieses erfolgt in „Defender" in einer Dimension und in „Laserbike" (siehe Anhang D) in 2 Dimensionen. Spiele, die dem Spieler den Blick aus einem Fenster vermitteln (Abb. 10.6), z.B. in einer Raumschlacht oder auf einem Rennkurs, arbeiten in 3 Dimensionen. Das Arbeiten in 3 Dimensionen ist sehr kompliziert. In diesem Buch werden nur 2 dimensionale Spiele behandelt, aber die Möglichkeit einer Erweiterung in die dritte Dimension ist immer vorhanden.

**Abbildung 10.6: Das Bildschirmfenster**

## SPIELE MIT MEHREREN BILDSCHIRMBILDERN

Ein Spiel mit mehreren Bildern auf dem Bildschirm ist leicht zu realisieren. Dabei ist der verfügbare Speicherplatz der wesentliche einschränkende Faktor. Jedes Bild muß irgendwo im Speicher zusammen mit allen Einzelheiten, Kniffen und Sonderregeln, die für den Bildschirm gelten, abgespeichert werden. Jede Bildschirmanzeige wird mit einer Kopierroutine in Gang gesetzt. Möglicherweise müssen mehrere Routinen kopiert oder bestätigt werden, um das Spiel in diesem Zustand zu halten. Ich will damit sagen, daß ein einzelner Bildschirm einen konstanten, über ihn treibenden Bombenhagel benötigen kann. Es müßte eine spezielle Routine geschrieben werden, die dafür sorgt. Das läuft darauf hinaus, daß jeder Bildschirm weit über 1K Speicherplatz benötigt, auch wenn er nur 20 Bildschirmzeilen belegt.

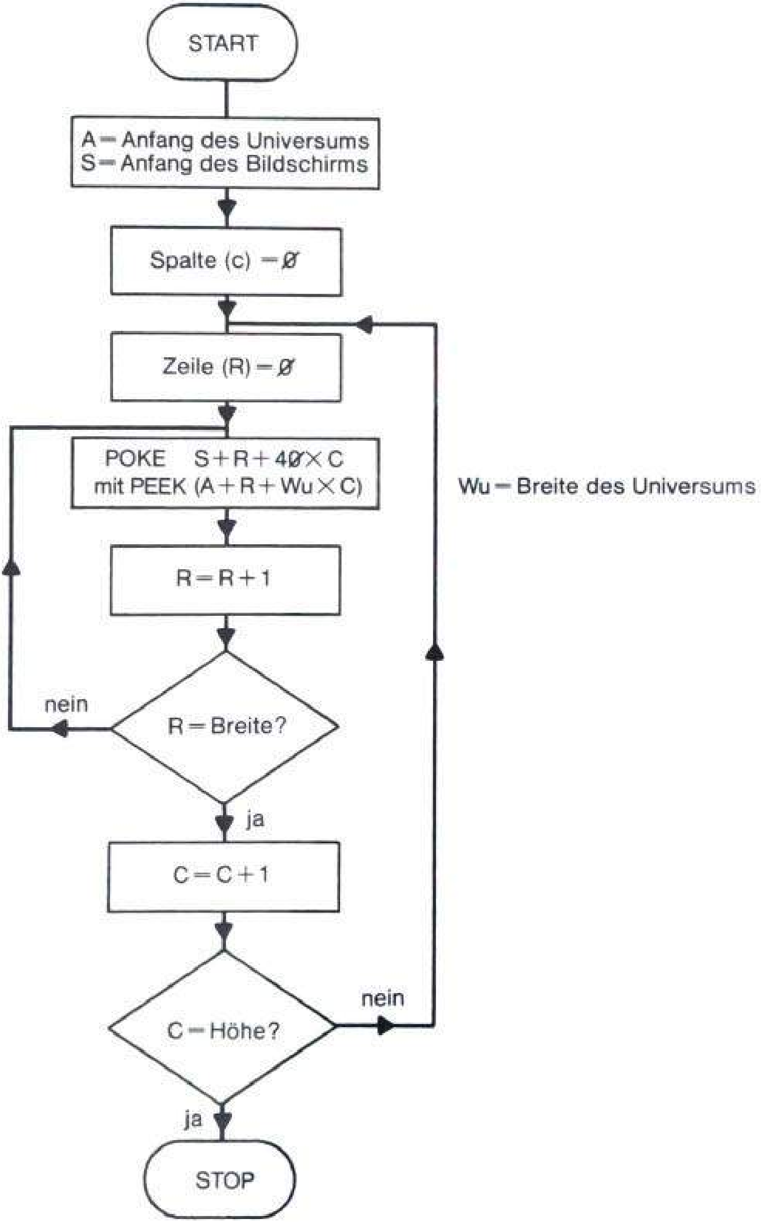
Der Rest des Speichers wird zur Durchführung von Verwaltungsaufgaben benötigt. Die Beschränkung des Speicherplatzes ist wahrscheinlich gar kein großes Problem, da es sowieso schwierig ist, sich mehr als ein paar originelle Bildschirme und Kniffe auszudenken.

Die Methode der Verwendung eines Fensters kann viel Spaß bereiten, da sich der Spieler leicht verirren kann, wenn Sie einen besonders großen Spielbereich wählen. Die erste Stufe ist die Zuweisung eines Speicherbereichs für „Welt", auf die das Fenster blickt. Dieser sollte mindestens 2K betragen. Dann entscheiden Sie, wie groß das Fenster sein soll. Ich bin der Ansicht, daß bereits ein halber Bildschirm zu groß ist. In „Laserbike" ist der Spieler auf ein 10 x 10 großes Blickfeld in eine 4K-Welt beschränkt. Als nächstes entscheiden Sie, wie die Welt aufgebaut sein soll. Die „Laserbike"-Welt ist zylindrisch – wenn Sie immer weiter nach rechts gehen, dann machen Sie eine Runde um sich selbst und erscheinen wieder links, aber eine Zeile tiefer. Das kommt daher, daß die Welt im Speicher wie ein kolossaler Bildschirm konfiguriert ist. Wenn Sie sich darüber hin bewegen, dann kommen Sie irgendwann zur Kante und werden um diese „herumgewickelt". Da wir jedoch den Blick aus dem Fenster haben, existiert die „Kante" als solche nicht. Dieser Effekt gilt nicht für die Ober- und Unterkante. Diese sind durch eine unüberquerbare Barriere abgesperrt. Um ein Herumwickeln zu vermeiden, müssen Sie auch eine senkrechte Barriere ziehen (diese verschlechtert das Spiel leicht). Jetzt können wir die Routine für die Erzeugung des Fenstereffekts schreiben.

Lassen Sie uns überlegen, welche Informationen wir haben. Wir wissen, wie groß der Spielbereich ist, und müssen diesen in 2 nicht zu unterschiedlichen Zahlen festlegen. Nehmen wir an, daß der Spielbereich 80 x 80 Bytes beträgt – insgesamt 6400 Bytes. Um sich vorzustellen, wie er gespeichert würde, betrachten Sie ihn als einen großen Bildschirm. Um nach links oder rechts zu gehen, würden Sie immer noch 1 addieren bzw. subtrahieren, aber die Bewegung nach oben und unten beträgt jetzt ± 80 Bytes. Wir kennen die Größe des Fensters, sagen wir 10 x 10 (das Fenster braucht nicht unbedingt ein Quadrat zu sein). Um jetzt den Abschnitt der Welt zu zeichnen, deren obere linke Ecke die Adresse 8000 besitzt, müssen wir 8000 – 8009 in aufeinanderfolgenden Adressen kopieren. Dabei beginnen wir mit der Bildschirmadresse, von der wir das Fenster projiziert haben wollen. Das ist natürlich die Adresse der oberen linken Fensterecke. Zum Zeichnen der nächsten Linie müssen wir zu 8000 zurückgehen und 80 hinzuzählen. Das bringt uns in der Welt eine Zeile nach unten. Wie zuvor kopieren wir jetzt von 8080 – 8089, beginnen aber bei der Fensteradresse +40. Wir machen dies einmal für jede Zeile, dann ist das Fenster projiziert. Abb. 10.7 zeigt, wie man es für ein Fenster mit den Abmessungen X, Y auf einer W Byte breiten Welt macht.

Das Programm „Laserbike" beruht auf dem Lichtradrennen von Tron. Als „Laserbike" zuerst in BASIC geschrieben wurde, erhielt man eine große Verzögerung durch das Löschen der Welt zwischen einer Runde und der nächsten. Natürlich dauert das Auffüllen eines 4K-Bereichs seine Zeit. Deshalb wurde die zweite

**Abbildung 10.7: Fensterprojektion**



Routine eingeführt, wodurch das Spiel zufriedenstellend wurde. Achten Sie jedoch darauf, daß die Welt selbst nur wenig Handlung enthält. Die Koordination von Handlung in BASIC, in einer Welt dieser Größe, würde das Spiel ruiniert haben. Deshalb gibt es keinen Feind, außer dem brennenden Weg, den Sie hinter sich lassen und den Bomben.

Die Verwendung eines Fensters ermöglicht die Realisierung ehrgeiziger Spiele. Z.B. arbeite ich gerade an einem Spiel, in dem ein Luftkissenboot, das mit Kanonen versehen ist, in einer See/Sumpf/Landzone patrouilliert, in der sich Gefahren wie Minenfelder, Kanonengruppen und Panzer befinden. Es wird dabei eine Fensterroutine verwendet, um ein Spiel über einen weiten Bereich zu ermöglichen, was zu einem Spiel mit schneller Handlung in Verbindung mit geschickter Steuerung des Luftkissenboots in schwierigen Windbedingungen führen sollte.

## ZWEIDIMENSIONALER FENSTERPROJEKTOR

Die folgende Routine erzeugt auf dem Bildschirm ein „Fenster", das aus einem größeren Speicherbereich kopiert wurde. Die Adresse der oberen linken Fensterecke im Speicher wird in 251–252 und die Adresse der oberen linken Bildschirmecke in 253–254 gespeichert. Diese werden während der Routine geändert und müssen deshalb zurückversetzt werden. Die Breite und Höhe des Fensters sowie die „Breite des Universums" sind variabel.

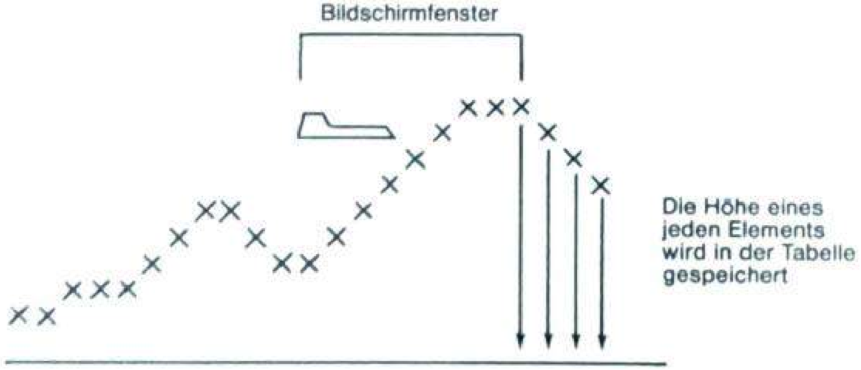
```
:LL0    LDX #[Fensterhöhe]
:LL1    LDY #0
:LL2    LDA (251),Y              Fensterkopieren
        STA (253),Y
        INY
        CPY #[Fensterbreite]
        BNE LL2
        CLC
        LDA 251
        ADC #[Breite des Universums] nächste Zeile
        STA 251
        LDA 252
        ADC #0
        STA 252
        CLC
        LDA 253
        ADC #40
        STA 253
```

```
LDA 254
ADC #0
STA 254
DEX
BNE LL1          Fertig?
RTS
```

## EIN LANDSCHAFTSFENSTER NACH „DEFENDER"-ART

Die Landschaft des Spieles „Defender" ist dadurch interessant, daß sie in beiden Richtungen rollt. Der Spieler ist auf einen bestimmten Bereich beschränkt – bewegt er sich immer weiter nach rechts, dann führt er eine volle Umrundung aus und taucht links wieder auf. Der Effekt ist, daß das Spiel auf einer Fläche gespielt wird, die sehr viel größer ist als die des Bildschirms. Die Frage ist, wie erzeugt man die Landschaft und wie weiß man, was beim Spiel sich wo befindet.

**Abbildung 10.8: Projektion einer Landschaft**




Eine Lösung ist die Speicherung der Landschaft als eine Liste von Höhen. Wenn Sie eine Liste mit 256 Eingaben wählen, dann erhalten Sie eine automatische, zyklische Adreßfolge. Der Spielbereich wird um einen Faktor von 6,4 vergrößert (siehe Abb. 10.8). Jedes Element auf dem Bild kann so betrachtet werden, als würde es sich eine bestimmte Anzahl von Zeilen über der Bildschirmunterkante befinden. Die gesamte Landschaft wird als eine Reihe von Höhen aufgelistet, die,

wenn sie geplottet werden, eine Landschaft bilden. Da Sie eine zyklische Adreßfolge verwenden, müssen Sie sicherstellen, daß die beiden Enden harmonisch ineinander übergehen.
Betrachten Sie jetzt das Schiff in Abb. 10.8. Angenommen, es befindet sich in Position X unserer Landschaft (0<X<255). Da es sich in der Mitte des Bildschirms befindet und wir das umgebende Gelände sehen wollen, müssen wir nur die Werte in der Tabelle in dem Bereich von beispielsweise (X–15) bis (X+15) plotten. Um zu erkennen, ob irgendwelche gegnerischen Schiffe zu zeichnen sind, müssen wir nur ihre Positionen daraufhin überprüfen, ob sie in dem gerade beschriebenen Bereich liegen. Diese Methode macht die Aufgabe einer „Defender"-Landschaft sehr leicht. Die Werte werden in einer 256-Byte langen Tabelle gespeichert, wobei eine Zero-Page-Adresse auf den Anfang der Tabelle zeigt. Verwenden Sie den indizierten X-Adressiermodus, dann haben Sie Zugang zu jeder Eintragung in der Tabelle. Obendrein erfolgt die zyklische Adressenvertauschung automatisch durch eine clevere Festlegung der Größe. Wir wollen uns nun das Programm zur Darstellung einer Landschaft ansehen:

```
       TXA              Das Schiff in Position X
       SBC #15          Beide Seiten des Schiffes plotten
       TAX
       LDY #30
:LOOP  LDA (TABLE,X)    Indirekter Zero-Page-Modus
       JSR PLOT         Schiff in den Koordinaten (Y,A) plotten
       INX
       DEY
       BNE LOOP
       RTS
```

Das Programm erklärt sich fast von selbst. Die Tabelle der Höhen beginnt bei TABLE (Zero-Page-Vektor). Die Routine wird eingegeben, dabei enthält das X-Register die Position des Schiffes. Das Programm zeichnet 30 verschiedene Geländeteile um das Schiff herum. DEY könnte durch einen komplizierteren Vergleich mit dem X-Register ersetzt werden, aber es ist sehr viel einfacher für die Darstellungs-(Plott)-Routine, wenn ihr Koordinaten auf dem Bildschirm mitgeteilt werden. Die Darstellungsroutine empfängt 2 Datenteile: die Anzahl der Zeilen, die man von der Bildschirmunterkante (A) hinaufzählen muß, und die Anzahl der Zeilen, die man darüberhinaus gehen muß (Y). Eine einfachere Arithmetik berechnet diese Positionen und plottet das notwendige Sinnbild. Vergessen Sie nicht, zuvor den Bildschirm zu löschen. Sonst überlagert eine Landschaft die nächste. Als ich diese Routine zum erstenmal schrieb, lief sie so schnell, daß ich die Landschaft nur sehen konnte, wenn ich das Schiff anhielt. Sie ist viel schneller als normales Rollen.

# FLOTTENBEWEGUNGEN

Es gibt nur wenige Spiele, in denen der Feind nicht aus einer Anzahl gleichartiger Fremdlinge besteht. (Und diese wenigen Spiele sind normalerweise ohnehin ziemlich langweilig.) Unser Problem ist, wie bewegt man eine gesamte Flotte von Schiffen in einer fließenden Formation, wie z. B. in „Galaxian". Die Antwort ist sehr einfach. Sie brauchen nur eine Routine, die eines von ihnen bewegt und verwenden dann diese Routine, um alle gleichartigen Fremdlinge zu bewegen. Schließlich sollten die Fremdlinge ohnehin alle identisch sein. Wenn Sie einen einzelnen Fremdling als eine normale Variable ansehen, dann ist eine Flotte von Ihnen einfach ein Array. Sie können sie einzeln bewegen, indem Sie einfach ein spezielles Element verändern, oder Sie können sie alle auf einmal beeinflussen. Scheuen Sie nicht davor zurück, Flotten in Ihren Spielen zu verwenden, da Einzelschiffspiele wirklich sehr langweilig werden können.

Es gibt im wesentlichen 2 Arten, Flottenbewegungen zu erzeugen. Die erste ist der Bildschirmansatz. Er besteht darin, daß Fremdlinge auf dem Bildschirm vorhanden sind und ein Aktualisierungsprogramm verwendet wird, das den Bildschirm abtastet und sie bewegt. Die Alternative dazu ist, alle ihre Positionen und Attribute in einer Tabelle abzuspeichern und eine getrennte Routine zu verwenden, die sie auf den Bildschirm plottet. Jede dieser Techniken hat ihre Vorteile, die ich detaillierter erläutern werde.

Das Schöne an der Bildschirmmethode ist, daß das, was auf dem Bildschirm existiert, identisch ist mit dem, was im Spiel vorhanden ist. Wenn Sie irgend etwas mit einer Bombe oder der Kanone vernichten, dann müssen Sie nur das entsprechende Bild auslöschen, um es aus dem Spiel zu nehmen. Beim Tabellensystem ist das Löschen nicht so einfach. Es kann einen rasend machen, wenn man einen Fremdling abschießt, ihn explodieren und verschwinden sieht und er in der nächsten Sekunde wiederbelebt wird. Die Bildschirmmethode hat eine wesentliche Schwachstelle. Wird eines der Schiffe in der Aufregung aus Versehen entfernt, so wird sein Vorhandensein nur im Gedächtnis des Spielers registriert, und vermittelt das Gefühl einer schlechten Kontinuität. Die Einführung neuer Gegenstände auf dem Bildschirm ist jedoch sehr einfach – Sie plotten sie einfach dahin, wo sie gebraucht werden. Werden Bildschirmmethoden angewendet, wie z. B. zum Rollen und Bombardieren (was ein besonderes Beispiel ist), dann muß das Abtasten in der Gegenrichtung zur Bewegung erfolgen. Andernfalls wissen Sie überhaupt nicht, was bereits aktualisiert worden ist und was nicht. Natürlich kann man dieses Problem auch dadurch umgehen, indem man das Zeichen von einer Form in eine andere umwandelt und dann wieder zurück.

Die Methode, die eine Tabelle benutzt, ist ein bißchen verzwickter, aber insgesamt gesehen die überlegenere Methode. Es passiert dabei folgendes: Im Speicher gibt es eine Adressentabelle aller unterschiedlichen Zeichen. Diese Tabelle ist in Wirklichkeit ein Maschinencode-Array und besitzt den Vorteil, daß mit ihrer Hilfe

**Abbildung 10.9: Aufeinanderfolgende Flottenbewegungen auf dem Bildschirm oder mit Hilfe einer Tabelle.**

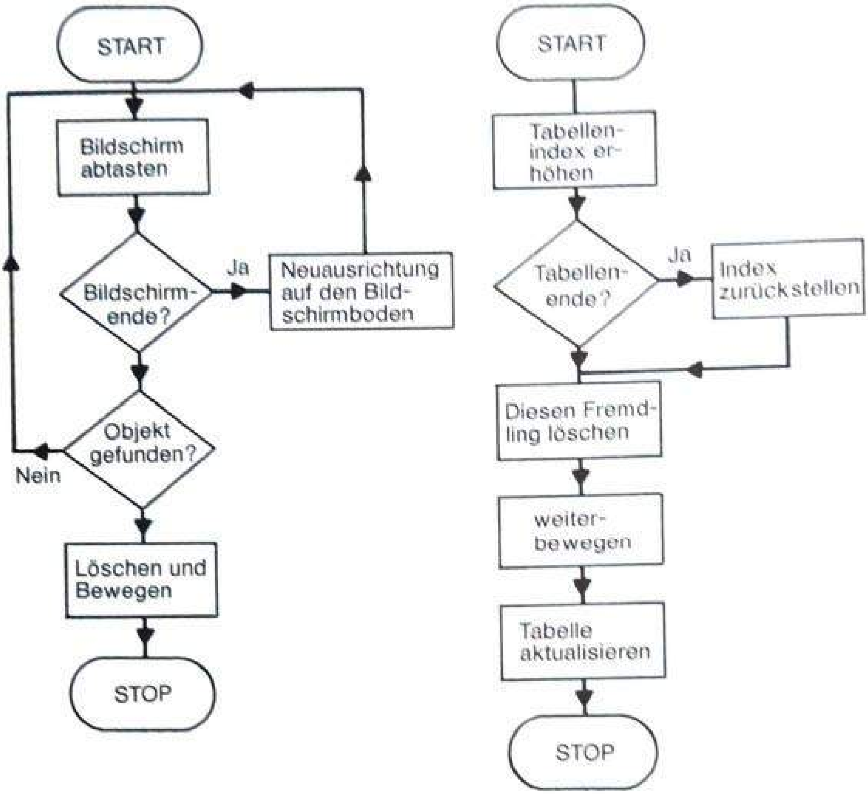



**Abbildung 10.10: Eine typische Fremdlingstabelle**

ein Datensatz in der Tabelle — nächster Datensatz

| | | | | | |
|---|---|---|---|---|---|
| nieder-wertiges Adreßbyte | höher-wertiges Adreßbyte | Vektor | Fremd-lingsart | nieder-wertiges Adreßbyte | höher-wertiges Adreßbyte |

Flottenbewegungen leicht koordiniert werden können. Die Zeichen müssen nicht auf jeweils ein Quadrat beschränkt werden, sie können sein, was immer Sie gerade möchten. Sie brauchen nur eine Routine, die ein Zeichen in einer beliebigen Adresse plottet, und schon haben Sie es geschafft. (Vielleicht möchten Sie meine alternative Spriteroutine aus dem vorausgegangenen Kapitel verwenden. Sie befaßte sich mit der Bewegung großer Zeichen, bestehend aus mehreren Rechtecken, über dem Bildschirm.) Die Anwendung eines derartigen Systems hat den Vorteil, daß man jedem Zeichen leicht einen Bewegungsvektor zuweisen kann, indem man einfach ein zweidimensionales und nicht ein eindimensionales Array verwendet. Das Aktualisierungsprogramm braucht dann nur noch die Tabelle durchlesen, die Vektoren heraussuchen und sie den Adressen hinzuzählen. Die Entfernung des vorherigen Zeichenbildes kann auch wünschenswert sein, damit das Zeichen keine Spuren hinterläßt. Die Verwendung dieser Vektormethode macht die Aufgabe leicht, Asteroiden in verschiedenen Richtungen über den Bildschirm zu bewegen.
Mit Hilfe beider Verfahren kann man die Geschwindigkeit der Feindbewegung gut steuern. Sie können ein oder zwei von ihnen zu einer Zeit, oder alle auf einmal bewegen. Ich persönlich bewege gerne ein paar von ihnen gleichzeitig, um den rieselnden Effekt von „Space Invaders" zu erzeugen. Abb. 10.9 zeigt die Methoden, die für beide beschriebenen Techniken verwendet werden. Die Anwendung einer Tabelle im Maschinencode sollte Ihnen keine Schwierigkeiten mehr bereiten. Für den Fall, daß es nicht so ist, beschreibe ich nachfolgend diesen Vorgang.
In der ersten Stufe ist eine Entscheidung über das Tabellenformat zu fällen. Für unsere Zwecke könnte sie aussehen wie die in der Tabelle 10.10: Sie besteht aus drei Abschnitten. Der erste enthält die Adresse, der zweite den Bewegungsvektor und der dritte die Fremdlingsklasse. Der letzte Tabellenabschnitt dient dazu, mehr als eine Fremdlingsart, zur gleichen Zeit innerhalb der gleichen Tabelle, steuern zu können. Sie verwendet 4 Bytes pro Eingabe. Deshalb müssen Sie nur 4 hinzuzählen, um von einer Eingabe zur nächsten zu kommen.
Die nächste Stufe ist, die Entscheidung zu treffen, wo die Tabelle hingesetzt werden soll. Legen Sie eine geeignete Stelle fest, z. B. von 49152 an, und siedeln Sie sie dort an. Zur Abtastung der Tabelle verwenden Sie eine 2-Byte-Adresse, die in einem Paar von Zero-Null-Speicherstellen enthalten ist, sowie den indirekten Zero-Page-X-Modus, um die Tabelle zu betrachten. Da die Zero-Page-Adresse auf das erste Byte der Eingabe zeigt, erhalten Sie die Adresse, den Vektor und den Code des Zeichens, indem Sie einfach X schrittweise erhöhen. Um zur nächsten Eingabe zu kommen, setzen Sie X auf 0 und addieren Sie 4 zur Speicherstelle.
Damit endet unsere Betrachtung der Bewegung einer großen Anzahl von Objekten. Es ist im Grunde ganz einfach. Ich möchte noch ein paar Punkte herausstreichen, bevor ich mit den Routinen weitermache. Haben Sie keine Angst davor, die Methoden zu mischen, indem Sie einen Bildschirmansatz für die Bewegung der Bomben und eine Tabelle zur Bewegung von Schiffen verwenden. Das ist eine

sehr befriedigende Methode zur Erstellung von Spielen und außerdem ist sie einfach. Beschränken Sie Ihre Tabellen nicht nur auf eine Eingabe (d.h. Adresse). Wie ich Ihnen bereits gezeigt habe, können Sie sehr viel größere Tabellen benutzen, die Ihren Zeichen weitere Attribute hinzufügen, wodurch sie realistischer wirken. Wenn Sie Geräusche verwenden, dann ist diese Stelle gut dazu geeignet, die Geräuschroutine aufzurufen, damit Sie sie mit der Bewegung synchronisieren können (oder auch gegen sie). Wenn Sie die Routine mit dem Interrupt verbinden wollen, dann lesen Sie den Abschnitt über die Verwendung des Interrupts am Anfang dieses Kapitels nach.

## ARRAYROUTINEN

Die folgende Routine liest eine Adressentabelle und plottet die Zeichen (so viele Bytes wie Sie möchten), die an diesen Adressen beginnen. Der Ausgangspunkt der Tabelle wird in den Speicherstellen 168–169 gespeichert. Diese werden nicht von der Routine geändert. Die Subroutine PLOT plottet das Zeichen an der in den Stellen 253–254 gefundenen Adresse. Für einfache Formen können Sie genauso gut Ihre eigene Routine schreiben, aber das alternative Sprite-System kann von Nutzen sein, wenn Sie größere Zeichen manipulieren wollen. Die Y-Vergleichsanweisung nimmt als Operanden die doppelte Anzahl der zu plottenden Zeichen (höchstens 227). Das geschieht deshalb, weil jedes Zeichen mit einer 2-Byte-Eingabe dargestellt wird.

```
:LL0    LDY #0
:LL1    LDA (168),Y
        STA 253
        INY
        LDA (168),Y
        INY
        STA 254
        TYA                 bewahrt den Wert von Y
        PHA
        JSR PLOT
        PLA                 holt ihn wieder hervor
        TAY
        CPY #[2 x Zeichenanzahl]
        BNE LL1
        RTS
```

Wenn Sie nur ein Zeichen plotten (z. B. ein "*"), dann sollte die PLOT-Routine etwa so aussehen:

```
LDA #[Zeichencode]
LDY #0
STA (253),Y
```

Die Routine ist schnell und wirkungsvoll. Das Löschen einer ganzen Flotte erreicht man durch Plotten in Code 32 (die Sprite-Routine macht das für Sie). Die Bewegung eines speziellen Zeichens erfolgt durch Addition eines Vektors zu einer Adresse der Tabelle. Die folgende Routine addiert den gleichen Vektor zu jedem Zeichen in der Tabelle. Wie oben erwähnt, wird die Adresse der Ausgangsposition der Tabelle in 168–169 gespeichert (und bleibt unverändert) und der Y-Vergleich nimmt die doppelte Zeichenanzahl als Operanden.

```
:LL0    LDY #0
:LL1    LDA (168),Y
        CLC
        ADC 251
        STA (168),Y
        INY
        LDA (168),Y
        ADC 252
        STA (168),Y
        INY
        CPY #[2 x Anzahl]
        BNE LL1
        RTS
```

Der Vektor ist in 251 und 252 enthalten. Zur Erzeugung einer Rückwärtsbewegung setzten Sie sehr große Zahlen in den Vektor ein (z. B. zieht 65535 von jeder Adresse 1 ab). Wird ein Schiff getroffen und Sie wollen es zerstören, so reicht es nicht aus, es nur vom Bildschirm zu löschen. Sie müssen es auch aus der Tabelle löschen. Die folgende Routine sucht die Tabelle nach Werten ab, die zu dem Wert der Adresse paßt, die in 251 und 252 enthalten ist. Der Wert des Y-Registers, der auf den passenden Wert zeigt, wird wieder in das Y-Register eingegeben. Die Routine speichert ihn außerdem an anderer Stelle zur zukünftigen Verwendung.

```
:LL0    LDY #0
:LL2    LDA (168),Y
        INY
        CMP 251
```

```
        BNE LL1
        LDA (168),Y
        INY
        CMP 252
        BNE LL2        Paßt?
        DEY            Ja, setzen Sie daher Y an den Anfang
        DEY
        STY 253        Speichern Sie es zur Sicherheit in 253
        RTS
:LL1    INY
        CLC
        BCC LL2
```

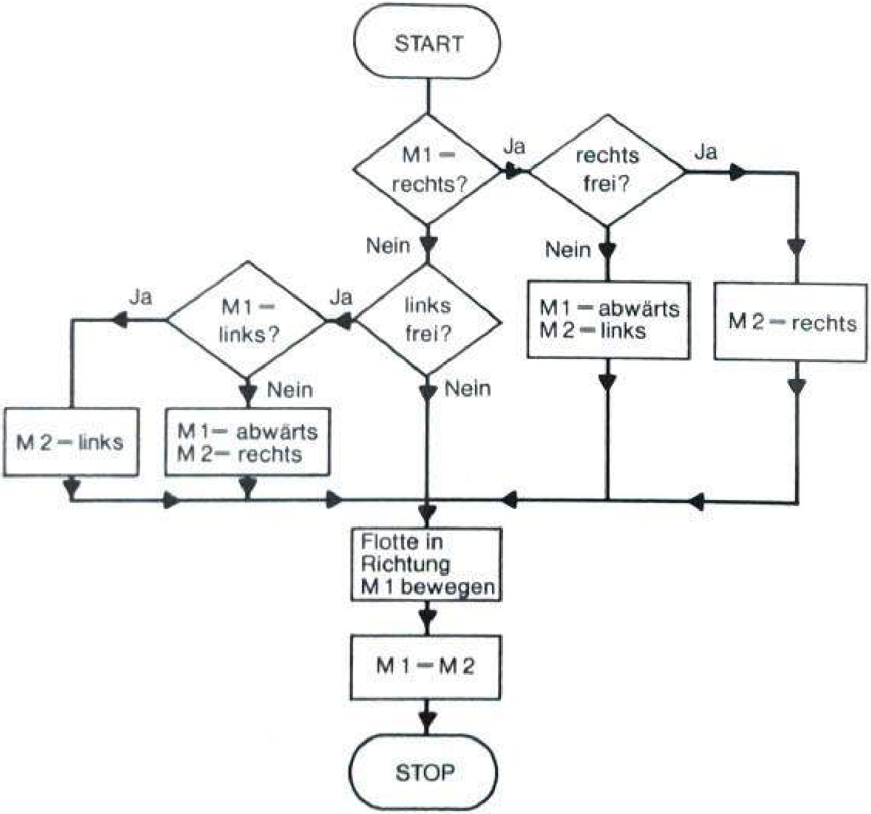
Wenn wir erst einmal wissen, welche Teile der Tabelle zu löschen sind, dann ist das Löschen selbst ganz einfach. Nehmen wir an, daß das X-Register doppelt soviel Elemente wie die Tabelle enthält (was wir immer weiter mit Y vergleichen). Nehmen wir außerdem an, daß das Y-Register den Index zur Tabelle enthält, der von der Suchroutine erzeugt wird, dann löscht die folgende Routine dieses Element. Das geschieht durch Überschreiben der Eingabe für das Löschen durch die letzte Eingabe in der Tabelle. Anschließend wird die letzte Eingabe durch Verringerung des Endzeigers beseitigt.

```
:LL0    DEX
        DEX
        STY 253
        TXA
        TAY
        LDA (168),Y
        LDY 253
        STA (168),Y
        INX
        INC 253
        TXA
        TAY
        LDA (168),Y
        LDY 253
        STA (168),Y
        RTS
```

Es hätte auch eine Schleife eingefügt werden können. Aber in dieser Weise ist es einfacher und deutlicher. Diese Routinen zur Handhabung von Tabellen sind für eine Zusammenarbeit ausgelegt: Sie verwenden häufig solche Dinge wie

Endzeiger, die das gleiche Datenstück untereinander herumreichen. Daß ich den unmittelbaren Modus bei den CPY-Anweisungen verwendet habe, bedeutet nicht, daß diese nur so arbeiten. Ändert sich die Größe der Tabelle von Zeit zu Zeit, dann ist dies sowieso nicht der richtige Ansatz. Die Löschroutine könnte innerhalb der Suchroutine untergebracht werden. Dadurch würde der Bedarf für den JSR-Aufruf beseitigt. Vergessen Sie aber nicht, zuerst das X-Register zu initialisieren. Diese Routinen sind „Module", aus denen Sie Ihre Bewegungsroutinen aufbauen. Sie können selbständig laufen, aber miteinander verbunden benötigen sie wahrscheinlich eine kleine Serviceroutine, um die Parameterübergabe zu synchronisieren.

**Abbildung 10.11: Vektorbestimmung für den Marsch feindlicher Flotten**



## DAS HIN-UND-HER-MARSCHIEREN

Viele Spiele besitzen feindliche Flotten, die sich durch eine Art „Hin-und-Her-Marschieren" annähern: sie bewegen sich nach links und rechts, bis sie die Kante erreichen, springen dann eine Zeile nach unten und bewegen sich in der Gegenrichtung. Die Logik für eine solche Bewegung ist nicht so leicht zu beschreiben. Abbildung 10.11 zeigt, wie der Bewegungsvektor für die Flotte gebildet wird. Sie rufen ihn einmal zur Flottenaktualisierung und nicht einmal für jedes Schiff auf. Es werden die beiden Variablen M1 und M2 verwendet. M1 enthält die erste und M2 die zweite Bewegung. Wenn Sie das Diagramm nachvollziehen, werden Sie erkennen, wie die Abwärtsbewegung und die Umkehrrichtung in die Reihenfolge der Ereignisse eingepaßt sind. Links und rechts frei bedeutet einfach, daß die Kante des Bildschirms auf Invasoren überprüft wird. Die Überprüfung sollte im Maschinencode erfolgen, obwohl die Flotten in BASIC durch Stringvariable unterstützt werden. Am besten machen Sie alles im Maschinencode, aber halten Sie es einfach.

## ZUFALLSZAHLEN

Das Problem der Erzeugung von Zufallszahlen im Maschinencode hat viele Möchte-Gern-Programmierer zum Aufgeben bewogen. Das Schlüsselproblem dabei ist, wie läßt man einen Computer, der eine Folge präziser Anweisungen befolgen kann, eine unvorhersagbare Zahlenfolge erzeugen? In BASIC gibt es dafür die Funktion RND(X). Im normalen Maschinencode gibt es dafür nichts. Der C 64 enthält aber einen Hardware-Zufallszahlengenerator! Denken Sie einen Augenblick über „weißes Rauschen" nach. Es ist nicht anders, als eine Reihe sich schnell ändernder Frequenzen. Die Änderung ist nicht nur schnell – sie ist zufällig. Die Ersteller des SID setzten eins der Register auf den Ausgang von Stimme 3. Befindet sich dieser dann im Modus „weißes Rauschen", dann können wir unsere Zufallszahlen durch PEEKen der Speicherstelle erhalten. Die Frequenz, mit der die Stimme arbeitet, bestimmt die Schnelligkeit, mit der die Zahlen erzeugt werden. Die Ausgangsfrequenz wird durch die folgende Funktion bestimmt:

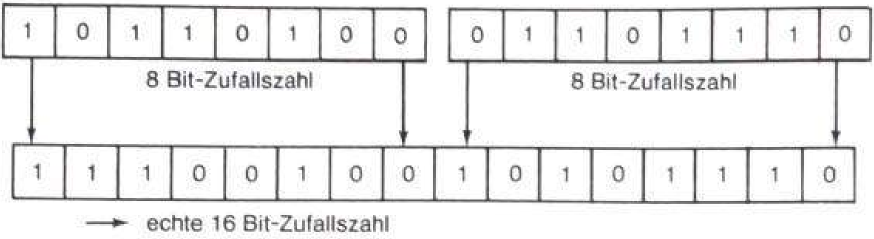
**f = (16 Bit-Frequenzeinstellung) x 0,0596 Hz**

Das bedeutet maximal 3906 Aktualisierungen pro Sekunde. Zur Nutzung des Generators müssen die folgenden Schritte unternommen werden:

1) **Schalten Sie Stimme 3 auf Rauschen.**
2) **Schalten Sie den Ausgang von Stimme 3 ab (setzen Sie Bit 3 von 54290).**
3) **Setzen Sie die Frequenz auf 54286 und 54287.**
4) **Verwenden Sie PEEK (54299) für Zufallszahlen.**

Wir erhalten Zufallszahlen im Bereich 0–255. Wie erzeugen wir eine zufällige Adresse? Setzen Sie einfach 2 Zufallszahlen aneinander. Indem wir immer nur ein Bit auf einmal nehmen und es einschieben, können wir Zahlen konstruieren, so groß wie wir wollen (Abbildung 10.12). Versuchen Sie jedoch nicht, zwei Zufallsbyte zusammenzuzählen, um eine Zufallszahl im Bereich 0–510 zu erhalten: Sie können sie aneinanderfügen, aber nicht addieren.

**Abbildung 10.12: Die richtige Additionsart**

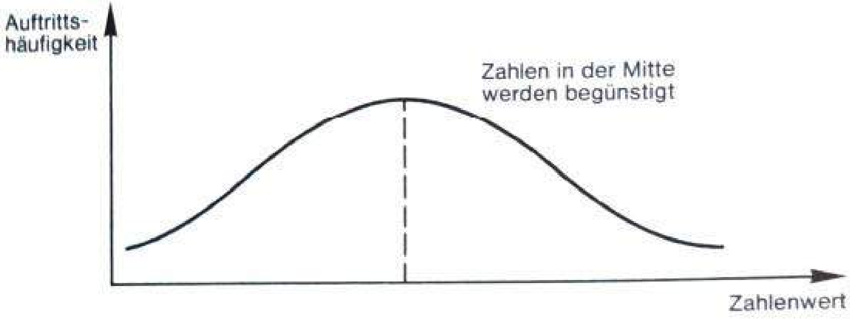



Das Problem bei der Addition ist, daß die Zahlen in der Mitte des Bereichs angehäuft werden. Werfen Sie zwei Würfel und schreiben Sie auf, wie oft jede Summe erscheint. Sie werden herausfinden, daß 7er viel öfter auftauchen als 12er und 2er zusammen. Das Ergebnis ist eine Glockenkurve, wie sie Abbildung 10.13 zeigt. Natürlich kann es sein, daß Sie solch eine ungleichmäßige Verteilung haben wollen. Je mehr Anteile Sie zusammenzählen, desto stärker wird die ungleiche Verteilung. Die Art, wie Sie Zufallszahlen erzeugen, ist weitgehend Ihnen überlassen. Durch ihren konstruktiven Einsatz kann man ein Spiel mit einem Element des Unbekannten würzen. Einzelheiten über ihre Verwendung siehe Flottenbewegungen und nichtlineare Bewegung in diesem Kapitel.

## NICHTLINEARE BEWEGUNG

Einige Spiele verlangen es geradezu, daß sich die Fremdlinge intelligenter bewegen, als einfache nur in einer Standardrichtung. Dazu kann es nötig sein, eine Zufälligkeitskomponente der Bewegung hinzuzufügen, oder daß die Fremdlinge gezwungen werden, einem vorgegebenen Pfad zu folgen, oder daß sie ein spezielles Objekt ansteuern müssen. Zweifellos bereiten die Spiele dadurch mehr Spaß, aber wie immer müssen Sie zuerst ein wenig Arbeit investieren, um dies zu erreichen. Um diese Technik transparent zu machen, werden einige der Beispiele in BASIC geschrieben, so daß Sie sie leichter nachvollziehen und eintippen können.

**Abbildung 10.13: Das Ergebnis konventioneller Addition: Eine ungleichmäßige Verteilung**




Beim ersten Fall, den wir betrachten wollen, wird dem Objektvektor eine zusätzliche Komponente hinzugefügt. Nehmen wir an, das Objekt sei eine Bombe, die den Bildschirm herabfällt. Ihr Vektor ist daher +40. Um etwas Leben in das Spiel zu bringen, können wir eine der Zahlen 1, 0 oder −1 zur Adresse sowie zum Vektor hinzuzählen. Die 0 läßt natürlich die Bombe unbewegt, während die 1 und die −1 sie nach rechts oder links ablenken. Es gibt nichts, was uns davon abhalten sollte, irgendeine Zahl aus dem gesamten Vektorbereich hinzuzuzählen. Das würde aber dazu führen, daß die Fortbewegung der Bomben zur Unterkante des Bildschirms hin verzögert wird. Die folgenden beiden Programme sollten dabei helfen, diesen Vorgang zu erläutern.

**Zufallsbombe**

```
10  REM ZUFALLSBOMBE
15  POKE 53281,1:PRINT"[SCHIRM NEU]":POKE 53281,0
20  S = 1044
30  DEF FNR(X)=INT(X*RND(1)+1
40  POKE S,42
50  S = S+42-FNR(3)
60  IF S>2023 THEN RUN
70  GOTO 40
```

**Zufallsbewegung**

```
10  REM ZUFALLSBEWEGUNG
20  POKE 53281,1:PRINT"[SCHIRM NEU]":POKE 53281,0
30  S = 1368
40  DEF FNR(X) = INT(X*RND(1)+1)
50  POKE S,46
60  S = S+2-FNR(3)+(2-FNR(3))*40
70  IF S<1024 OR S>2023 THEN RUN
80  POKE S,160
90  GOTO 50
```

Natürlich wird eine echte Zufallsbewegung nur durch Addition einer Zahl aus dem gesamten Vektorbereich zur Adresse erreicht. Der Vorgang, eine zufällige Zahl aus einer Tabelle im Maschinencode herauszulesen, wird nachfolgend beschrieben. Die erste Stufe ist die Erzeugung einer Zufallszahl. Man erhöht sie durch PEEKen von 54299 (siehe Abschnitt Zufallszahlen in diesem Kapitel). Jede Eintragung in der Tabelle besteht aus 2 Bytes (deshalb können wir negative Zahlen als große Zahlen darstellen). Da es 8 mögliche Richtungen gibt, müssen wir erreichen, daß die Zufallszahl einen der folgenden Werte annimmt: 0, 2, 4, 6, 8, 10, 12, 14. Um diesen Bereich zu erhalten, verknüpfen wir einfach unsere Zufallszahl über die Funktion AND mit 14. Ein schneller TRX-Befehl schiebt den Wert in das X-Register, und wir können den eingetragenen Wert unter Verwendung des ABSOLUT-X-Adressiermodus heraussuchen.

```
LDA 54299
AND #14
TAX
```

LDA TABLE,X ergibt die erste Eintragung und INX,
LDA TABLE,X liefert die zweite. Dieser Vektor wird einfach der zweiten Adresse hinzuaddiert.

## NICHTZUFÄLLIGE BEWEGUNG

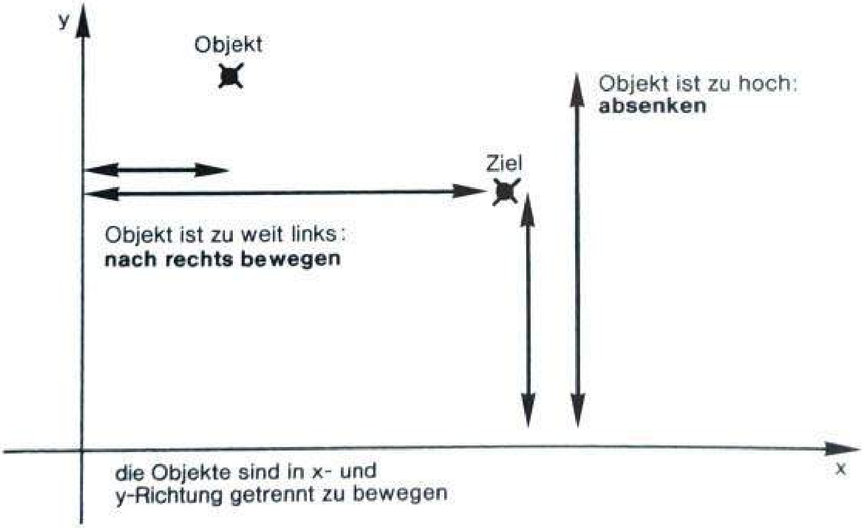
Die natürliche Weiterentwicklung der Zufallsbewegung ist eine festgelegte Bewegung. Was ich damit meine, ist, Sie möchten vielleicht, daß ein Fremdlingsschiff einen spiralförmigen Weg bis zur Unterkante des Bildschirms nimmt und nach dem Raubzug wieder zurückkehrt. Anstatt zu versuchen, einen Algorithmus aufzustellen, der die nächste Position berechnet, ist es leichter, nacheinander Zahlen aus einer Tabelle hinzuzuzählen. Die Struktur ist einfach: Sie benötigen nur einen Zeiger, der Ihre Position in der Tabelle kennzeichnet, Bewegen Sie dabei das

Objekt einfach dadurch weiter, indem Sie den Zeiger eine Stelle weiterrücken und den entsprechenden Vektor aus der Tabelle heraussuchen. Ein Null-Vektor zeigt das Ende der Tabelle an und könnte bewirken, daß der Zeiger auf eine andere Position in der Tabelle zurückgestellt wird. Diese Technik wird im nächsten Programm illustriert, das die Vektortabelle in Form von DATA-Anweisungen enthält.

**Tanz**

```
10  REM **TANZ
20  DATA 41,41,41,41,41,41,41,41,41,41,41,41,41,41,41,1,1,1,1,1
30  DATA -39,-39,-39,-39,-41,-41,-41,-41,-41
40  DATA -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,41, 39,39,39,39
50  DATA 1,1,1,1,1,1,1,1,1,0
60  DIM P(60)
70  L=1
80  READ P(L):IF P(I) <> 0 THEN L=L+1:GOTO 60
90  POKE 53281,1:PRINT"[SCHIRM NEU]":POKE 53281,0
100 S=1024:L=1
110 POKE S,42:L=L+1:POKE S+P(L),42:POKE S,46
120 IF P(L)=0 THEN L=10
130 IF S>2000 THEN END
140 S=S+P(L):GOTO 110
```

**Abbildung 10.14: Theorie der Zielansteuerung**



Zunächst wird diese Tabelle in ein Array eingelesen und die Objektposition initialisiert. Die Tabelle wird dann abgearbeitet, bis die Nulleintragung erreicht wird. Danach wird der Zeiger auf eine neue Position innerhalb der Tabelle zurückgesetzt. Das Programm macht so endlos weiter, mit dem Ergebnis, daß ein Objekt von der oberen linken Ecke herabflattert und dabei anfängt, eine Tanzfigur aus 8 Schritten auszuführen. Diese Methode kann leicht für die Anwendung im Maschinencode angepaßt werden.

## ZIELANSTEUERNDE BEWEGUNG

Die natürliche Weiterentwicklung der vorbestimmten Bewegung ist die zielansteuernde Bewegung. Dabei steuert ein Objekt ein anderes Objekt an. Das klassische Spiel „Zombie Island" verwendet dieses Prinzip mit einer guten Wirkung. Der Mechanismus, der diesen Vorgang bewirkt, ist ebenfalls bemerkenswert einfach. Die erste Stufe ist das Aufbrechen der Adressen in ein Paar von X, Y-Koordinaten, wie man sie in Graphiken verwendet. Aus diesem Grund sind Sprites am leichtesten zu verwenden, da sie bereits in dieser Weise ausgedrückt werden. Betrachten Sie jetzt Abb. 10.14, die zwei Punkte a und b zeigt. b ist der Punkt, auf den sich a zubewegt. Zur Aktualisierung der Position von a wenden Sie die folgenden Regeln an:

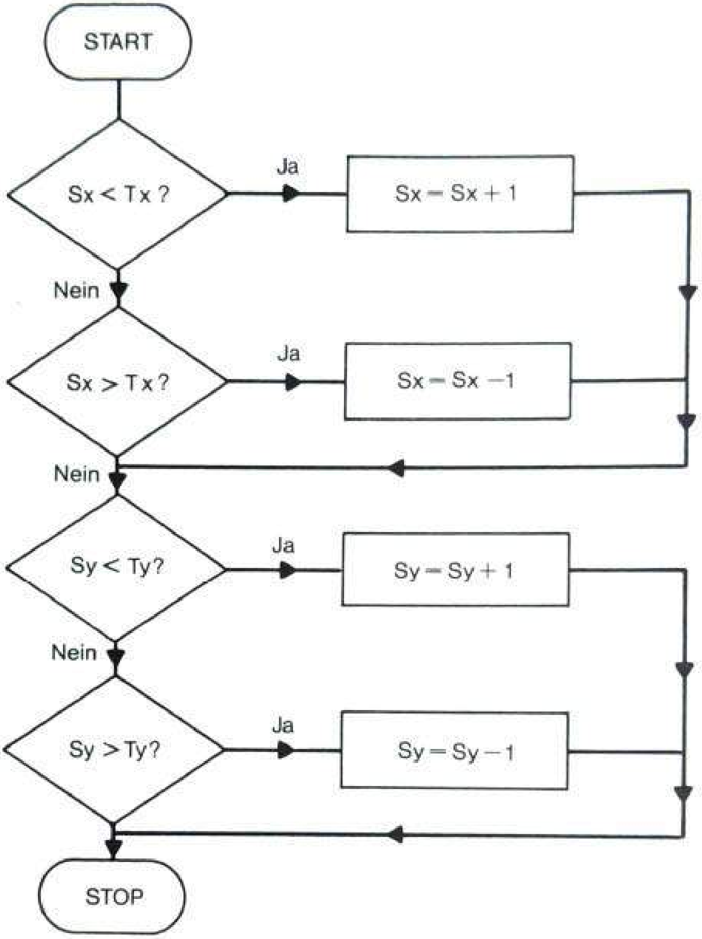
**Ist $X_b > X_a$, dann ist $X_a$ zu erhöhen.**
**Ist $X_b < X_a$, dann ist $X_a$ zu verkleinern.**
**Ist $Y_b > Y_a$, dann ist $Y_a$ zu erhöhen.**
**Ist $Y_b < Y_a$, dann ist $Y_a$ zu verringern.**

Wenn Sie sich jetzt Abb. 10.15 ansehen, sollten Ihnen die Gründe für diese Regel deutlich werden. Das BASIC-Programm aktualisiert die Sprites 1 bis 7 in Richtung auf den Standort des Sprites 0.

```
10     REM **SPRITE ANSTEUERUNGSROUTINE
20     HX=PEEK(53248):HY=PEEK(53249)
30     FOR A=53250 TO 53262 STEP 2
40     X=PEEK(A)+SGN(HX-PEEK(A))
50     Y=PEEK(A+1)+SGN(HY-PEEK(A+1))
60     POKEA,Y:POKEA+1,Y
70     NEXT:GOTO 30

1000 REM **AUFSTELLEN VON SPRITES ZUR DEMONSTRATION
1010 FOR A=53248 TO A+15:POKE A,FNR(255):NEXT
1020 POKE 53269,255
1030 FOR A=2040 TO 2047:POKE A,0:NEXT
     ----FNR(X)=INT(X*RND(1)+1)----
```

**Abbildung 10.15: Ansteuerung von Sx, Sy) auf (Tx, Ty)**




Dieses Programm treibt die Sprites 1 bis 7 in Richtung des Sprites 0. Es ist sehr langsam, aber das Programm zeigt deutlich, wie der Algorithmus arbeitet. Die Logik befindet sich in den Zeilen 40 und 50, die die vorhandene Funktion SGN(X) verwenden. Diese ergibt entweder −1, 0 oder 1, je nachdem, ob die Zahl kleiner als 0, 0 oder größer als 0 ist.
Lassen Sie zuerst die Subroutine 1000 laufen, um einige Sprites für eine Jagd aufzustellen.

Das folgende Maschinencodeprogramm tut das gleiche, aber sehr viel schneller. Diese Routine hängt man am besten an den Interrupt an. Siehe hierzu den Abschnitt über den Interrupt, weiter vorne in diesem Kapitel. Sie können den gleichen Effekt erreichen, wenn Sie normale Zeichen verwenden, aber damit ist es sehr viel langweiliger. 8 Sprites sollten genügen. Versuchen Sie zu vermeiden, alle Ihre Sprites für andere Zwecke zu verbrauchen.

Die Verwendung nichtlinearer Methoden belebt Ihr Spiel, übertreiben Sie es aber nicht. Die hier skizzierten Methoden sind nicht die einzigen, die es gibt, es sind nur die einfachsten. Zögern Sie deshalb nicht, Ihre eigenen, maßgeschneiderten Routinen zu schreiben. Um z. B. der Spriteroutine ein paar Einzelheiten hinzuzufügen, können Sie sie so ändern, daß ein Sprite automatisch abgeschaltet wird, wenn er am „Ziel" ankommt. Diese Spriteroutine bildet fast schon alleine ein Spiel. Der Spieler steuert Sprite 0 mit einem Joystick und muß versuchen zu vermeiden, daß er von irgendeinem anderen Sprite getroffen wird.

Die Logik ist ein wenig kompliziert, aber wenn Sie sie in Gedanken durchlaufen, werden Sie herausfinden, was im einzelnen passiert. Das Y-Register ist abwechselnd 1 und 0, so daß die Anweisung SBC 53248,Y abwechselnd die X- und Y-Koordinaten vergleicht. Die beiden DEC-Anweisungen und der nachfolgende INC-Befehl bewirkt einfach die Erzeugung einer DEC-Anweisung. Wenn Sie den Pfeilen folgen, sollten Sie erkennen, was passiert.

```
:LL0    LDY #1
        LDX #14
:LL1    LDA 53249,X
        SEC
        SBC 53248,Y     Vergleichen
        BEQ LL2         gleich?
        BCC LL3         negativ?
        DEC 53249,X     dann ist es positiv
        DEC 53249,X
:LL3    INC 53249,X
:LL2    TYA             Y hin- und herschalten,
        EOR #1          damit wir die X- und Y-Koordinaten
        TAY             Y-abwechselnd vergleichen
        DEX
        BNE LL1
        RTS
```

Diese Routine läuft in weniger als einer 5/1000 Sekunde ab und erzeugt deshalb eine wirklich schnelle Anzeige. Wenn man sie an den Interrupt anhängt, kann es sehr viel Freude machen. Das ermöglicht Ihnen auch ein Spiel „Flüchten und

Entrinnen" in BASIC zu schreiben. Einzelheiten über das Entfernen kollidierender Sprites sind im Abschnitt über Sprites, weiter vorne in diesem Kapitel, enthalten.

## DER FARBSPEICHER

Über den Farbspeicher gibt es eine ganze Menge zu berichten. Für jedes Zeichen auf dem Bildschirm gibt es ein entsprechendes Farbbyte, das die Farbe des Zeichens festlegt. Die in 53281 festgelegte Schriftfeldfarbe steuert die Farbe des Hintergrundes. Man kann tolle Sachen damit machen. Eine mögliche Anwendung ist die Idee des „plötzlichen Auftauchens". Dafür stellt man die Schriftfeldfarbe als Farbe in den Farbspeicher. Damit kann man etwas auf dem Bildschirm anzeigen, was zwar vorhanden, aber nicht sichtbar ist. Durch Änderung der Hintergrundfarbe springen die „versteckten" Bildschirmzeichen plötzlich hervor.
Werden unterschiedliche Werte in verschiedene Abschnitte des Farbspeichers gesetzt, dann erhalten wir einen farbigen Bildschirm. Die alten „Space Invaders" Maschinen haben waagerechte Farbstreifen, die durch Aufkleben von grünem und rotem Plastikklebeband auf den Bildschirmen erzeugt werden. Dieser Effekt kann sehr leicht simuliert werden und einen bestimmten Zweck in einem Spiel erfüllen (z.B. einem Spieler mitteilen, daß er sich in einer gefährlichen Zone befindet – ROT). Wird der Farbspeicher am Anfang eines jeden Spieles eingestellt, dann können Sie ihn für den Rest der Spieldauer vergessen. Andererseits können Sie auch die Farbe POKEn, während Sie die Fremdlinge herumbewegen. Dadurch würde aber das Programm kompliziert, wo es einfach sein kann.
Haben nur ein paar Bereiche des Bildschirms die gleiche Farbe wie der Hintergrund, dann sind diese von der Wirkung her „unsichtbare" Flecken. Ihr Inneres ist ein unbekanntes Territorium, wenn der Spieler sie erforscht, wird der Inhalt vielleicht entdeckt. Tut man dies, dann spart man einen weiteren Speicherabschnitt, der das enthält, was auf dem Bildschirm sein sollte. Um es noch einmal zu sagen, wenn man so vorgeht, dann sind bestimmte Aufgaben sehr viel leichter zu lösen.
Der Randfarbenzeiger (53280) ist eine weitere wertvolle Speicherstelle. Seltsamerweise hat die Randfarbe eine Wirkung auf die Hintergrundfarbe. So wirkt das Schriftfeld umso dunkler, je heller der Rand ist. Das ergibt weitere 16 Farbstufen pro Hintergrundfarbe! Ich persönlich verwende den Rand als Informationsquelle für den Spieler oder auch nur als Führung, die anzeigt, in welcher Ebene der Spieler gerade spielt. Wenn Sie die Hintergrund- und Randfarben bei jeder Angriffswelle ändern, dann trägt das mit dazu bei, das Spiel frisch und aufregend zu halten.

# ANHANG A

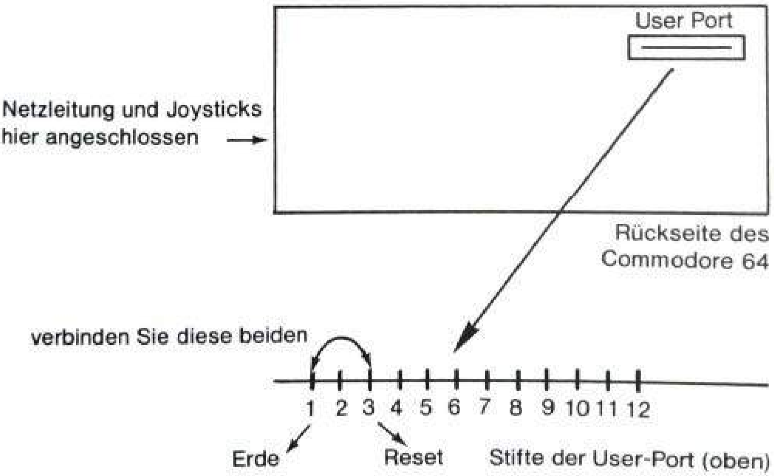
# HILFSPROGRAMME

Die Anhänge enthalten eine Menge Informationen, die nicht so recht an anderer Stelle in dieses Buch hineinpassen wollten. Es sind Hilfsprogramme, Routinen und Einzelheiten über die Maschine, die alle eine Hilfe für Sie sein sollten.

## WIEDERANLAUF NACH ABSTURZ

Es wird früher oder später passieren – Ihr Programm blockiert und die Tastatur ist tot. (Es ist ein wunderschönes Gefühl, wenn 5 Stunden Arbeit verloren sind ...) Als erstes sollten Sie es sich zur Gewohnheit machen, mindestens alle halbe Stunde einen SAVE-Befehl einzugeben.
Aber sogar bei einem Absturz ist, auch wenn die Kombination RUN/STOP + RESTORE nicht anspricht, noch nicht alles verloren (wenn Sie RUN/STOP + RESTORE abschalten wollen, dann müssen Sie nur den Vektor 792–793 ändern). Auf der Rückseite der Maschine finden Sie die in Abb. A. 1 gezeigten Anschlußstellen. Nehmen Sie eine leitende Pinzette und schließen Sie, wie es gezeigt wird, den RESET-Stift eine halbe Sekunde lang kurz. Lassen Sie ihn danach los. Der Bildschirm wird gelöscht und der Commodoretitel leuchtet auf. Tippen Sie jetzt die

**Abbildung A.1: Erdung des Reset-Stiftes**



folgenden Zahlenfolge z.B. in den Bandpuffer (jede andere Stelle ist genauso gut) und rufen Sie sie mit dem entsprechenden SYS-Befehl auf.

169,1,141,2,8,32,51,165,24,165,34,105,2,133,45,165,35,105,0,133,46,76,94,166

Treten Sie zurück und beten Sie. Ihr Programm ist jetzt wieder auferstanden. Natürlich laden die Cleveren unter uns diese Routine am Anfang eines jeden Tages irgendwo in eine sichere Stelle. Beim ersten Absturz brauchen Sie dann einfach nur noch einen schnellen SYS-Aufruf.

```
LDA #1
STA  2050        Programmwiederbeleben
JSR  42291
LDA  34          Hilfsprogrammzeiger
CLC
ADC #2
STA  45
LDA  35
ADC #0
STA  46          Holt die Variablen hervor
JMP  42590       Ende
```

Wenn Sie diese Wiederaufnahmeroutine jederzeit resident halten wollen, dann müssen Sie sich einen anderen Ort als den Bandpuffer suchen, da dieser bei RESET gelöscht wird. Versuchen Sie es in dem freien 4K RAM-Bereich.

## UNTERBRINGUNG DES PROGRAMMS UND SPEICHERVERWALTUNG

Wenn Sie ein BASIC-Programm mit Maschinencodeunterstützung, UDG- und Spritedaten und großen Bildschirmentwürfen laufen lassen wollen, dann tritt das Problem auf, wo soll man die Routinen unterbringen, und wo kann man die Maschinencodevariablen, -parameter und -tabellen speichern. Machen Sie sich darüber keine Sorgen, es ist eine 64K-Maschine. Wenn Sie es fertigbringen sollten, den Speicher mit einem vernünftigen Code zu belegen und immer noch mehr Platz benötigen, dann sind Sie entweder verrückt oder Sie benötigen eine größere Maschine.
Traditionell ist bei Commodoremaschinen der Platz für die Maschinencoderoutinen der Bandpuffer. Wir haben ihn immer noch und er reicht von 828 bis 1019, aber auf beiden Seiten davon erstreckt sich ein unbenutzter Speicherbereich von

820 bis 1023, also 204 Bytes. Dieser Bereich kann den größten Teil der Sammlungen von „unterstützenden" Routinen aufnehmen. Wenn Sie aber wirklich in Druck sind, dann sind 4K von 49152 bis 53247 verfügbar. Sprites können sehr gut im Bandpuffer untergebracht werden (es gibt nicht viele andere Stellen, in die sie gut hineingehen). Deshalb können Sie gezwungen sein, sich der 4K-Zone zu bedienen. Dadurch wird die Ausführungsgeschwindigkeit überhaupt nicht beeinflußt.

Das Programmierhandbuch hat die Angewohnheit, die Mitte des BASIC-Arbeitsraums für solche Dinge, wie UDG-Tabellen, zu verwenden. Das erscheint wenig sinnvoll und ist es auch nicht. Sie können aber das Problem umgehen, indem Sie den Arbeitsraum selbst verschieben. Das ist natürlich der Schlüssel, um wirklich große Programme unterzubringen. Die Speicherstellen 43 und 44 enthalten den Anfang des Arbeitsraumes. PEEKen Sie diese und Sie erhalten die Adresse 2049. 55 und 56 enthalten die höchste von BASIC verwendeten Adresse – den Wert 40960. Versuchen Sie, das obere Ende so weit nach unten zu drücken, daß es nicht mehr als 8K vom Anfang entfernt ist. Sie haben jetzt ausreichend Platz für ein BASIC-Programm und mehr als 30000 Bytes zum Herumspielen im Maschinencode. Zwei wichtige Punkte sollten Sie beachten: Erstens, POKEn Sie das obere Ende nach unten, bevor Sie irgendwelche Variablen erzeugen, und geben Sie dann CLR ein, um den C64 zu informieren. Zweitens, POKEn Sie das untere Ende nur außerhalb eines Programms und rufen Sie dann CLR auf (wenn Sie das innerhalb machen, verlieren Sie die Grundlage Ihres Programms).

Dadurch sollten alle Probleme der Unterbringung von Routinen gelöst sein. Die Verfügbarkeit des Zero-Page-Speichers ist jedoch eine andere Sache. Die Zero-Page ist etwas ganz besonderes und beim C64 ist sehr wenig Platz übrig. Wenn Sie natürlich kein BASIC verwenden, dann steht Ihnen alles zur Verfügung (außer 0 und 1). Die Speicherstellen 2, 251, 252, 253 und 254 sind auf jeden Fall für den Benutzer frei. Das ist ausreichend. Sollten Sie aber wirklich einmal in Druck kommen; einige der anderen Speicherstellen werden so selten verwendet (insbesondere die Bandregister), daß Sie sie ungestraft verwenden können. Ein weiteres brauchbares kleines Loch befindet sich in 679 bis 767. Seien Sie jedoch sehr vorsichtig mit Zero-Page-Operationen – sie können Sie Ihr Programm kosten.

## ÜBERGANG VON BASIC ZUM MASCHINENCODE

Dieser Abschnitt mag überflüssig erscheinen, aber wahrscheinlich kennen Sie nur ein oder zwei der möglichen Wege, in den Maschinencode hineinzukommen. Es gibt dafür 5 Hauptwege. Jeder hat seine Vor- und Nachteile. Deshalb ist es wert, sie näher zu betrachten, anstatt einfach nur blind ein SYS zu verwenden.

Der SYS-Aufruf ist der Standardeinstieg. Sie verwenden einfach SYS(X) und rufen so die Maschinencodesubroutine in X auf, als wenn es ein GOSUB-Befehl wäre. Diese Methode ist einfach und schnell, aber die Übergabe von Daten in und aus einer Routine kann auf der BASIC-Seite schwierig werden.

Dieses Problem wird durch die Verwendung des wenig benutzten Befehls USR(X) überwunden. Dieser Befehl wird wie eine Funktion, z.B. SIN(X) oder COS(X) behandelt. Der Wert in Klammern wird ausgewertet und in den „Gleitkommaakkumulator 1" gesetzt. Die zugehörige Maschinencoderoutine wird anschließend aufgerufen. Die Adresse wird vorher zugeordnet, indem man sie in den USR-Vektor in 785 und 786 setzt. Nach Beendigung der Routine wird der Wert des Gleitkommaakkumulators als Wert von USR(X) ausgegeben. Deshalb wird die Routine wie folgt aufgerufen:

```
A=USR(1000)
```

Einzelheiten über den Gleitkommaakkumulator entnehmen Sie bitte Anhang C (Text- und Variablenspeicherung in BASIC). Er besteht aus 5 Bytes und kann deshalb für die Übertragung einer ganzen Menge Daten von und zu den Maschinencoderoutinen verwendet werden. Durch Änderung des USR-Vektors kann eine Anzahl von verschiedenen Routinen während des Programms aufgerufen werden. Wandeln Sie ein Programm einer früheren Commodore-Maschine um, dann achten Sie auf jeden Fall darauf, daß Sie die Vektorzeiger von 1 und 2 auf 785 und 786 ändern.

Der Interrupt liefert eine weitere Verbindung zum Maschinencode. Sie nehmen einfach den Vektor von 788 und 789 und richten ihn auf Ihre Routine. Dabei müssen Sie besonders darauf achten, daß das Ende Ihrer Routine mit dem Anfang des Interrupt-Serviceprogramms in 59953 verkettet wird. Tun Sie dies nicht und hören mit einem RTI auf, dann werden Sie alle Ihre Verwaltungsroutinen verlieren. Der Einstieg über den IRQ-Interrupt benötigt ein bißchen mehr Vorausplanung bei der Programmierung, aber er ist die Mühe wert. Vergessen Sie nicht, daß die Uhr (TI$) völlig außer Takt gerät, (sich verlangsamt) wenn die Ausführung Ihrer Routine mehr als 1/60 Sekunde dauert. Durch Anwendung von einem der CIA können Sie sogar Ihren eigenen Interruptsignalplan vorgeben. Zweifellos ist der Interrupt das wesentliche Hilfsmittel zur Zügelung des C64.

Eine hinterlistige Weise, sich in den Maschinencode einzuschleichen, ist das Abfangen des BASIC-Interpreters, während dieser fröhlich eine Zeile abtastet, um zu sehen, was er sagt. Die als CHRGET bekannte Routine liefert mit jedem Abruf aufeinanderfolgende Zeichen einer Zeile. Die Routine ist aufmerksamerweise im Zero-Page-RAM vorhanden. Das bedeutet, daß wir einen „Keil" hineintreiben und so die hintere Hälfte für eine Routine Ihrer Wahl abzweigen können. Die Möglichkeit, in BASIC einen neuen Befehl hinzuzufügen, wird durch Ausschauhalten nach einem Steuerzeichen eröffnet. Sehen Sie sich die CHRGET-Routine an:

```
INCZ122
BNE↑2
INCZ123
LDA£-----
CMP#58
BCS↑10
CMP#32
BEQ↑239
SEC
SBC#48
SEC
SBC#208
RTS
```

Der Trick dabei ist, die erste SEC-Anweisung durch ein JMP$820 (oder irgendwo anders hin) zu ersetzen. Dann können wir bei 820 folgendes machen:

```
CMP #0
BEQ ↑ [Benutzerroutine]
SEC
SPC#48
usw., usw.
```

Das einzig echte Problem ist, wie man die JMP-Anweisung einsetzt, ohne einen Fehler zu verursachen (da nicht alles auf einmal gemacht werden kann). Die Antwort ist, ein gesondertes Codestück zu verwenden, um den Keil hineinzutreiben. Wie der Interrupt ist diese Methode handlich, aber verzwickt.
Wenn Sie den Vektor z.B. in Richtung auf eine LIST- oder SAVE-Routine ändern, dann bewirkt dieser Befehl einen Sprung zum neuen Vektor. Vielleicht wollen Sie die Wirkung eines alten Befehls modifizieren oder seinen Sinn vollständig ändern. Suchen Sie sich einige Vektoradressen aus der Tabelle, mit denen Sie herumspielen möchten. Dieses ist die letzte der 5 Hauptarten zum Eintritt in den Maschinencode. Es gibt noch mehr davon. Sie sind aber nicht der Mühe wert, hier aufgelistet zu werden.

## BENUTZERDEFINIERBARE GRAPHIK (UDG)

Dieser Abschnitt ist dem Thema der UDGs gewidmet und enthält einen kleinen Editor. UDGs sind einfach graphische Zeichen, die der Benutzer entworfen hat. Wenn Sie ihn einschalten, dann besitzt der Commodore 64 einen Satz aus 256

verschiedenen vordefinierten Zeichen. Diese sind zwar bemerkenswert vielseitig, aber es ist oft besser, Ihre eigenen zu definieren. Sie müssen nur die Form Ihres neuen Zeichens aufzeichnen, ein quadratisches Gitter darüberlegen (8+8) und entscheiden, welche Quadrate gefüllt werden sollten. Dann addieren Sie einfach jede Zeile, wobei Sie sie als 8-Bit-Binärzahl betrachten, und erhalten so 8 Zahlen. Damit ist die Definition komplett.
Das Problem dabei ist, der Standardzeichensatz ist im ROM gespeichert, sodaß wir ihn nicht direkt verändern können. Stattdessen müssen wir ihn irgendwo anders hinkopieren, die entsprechenden Matrizen ändern und der Maschine mitteilen, daß sie den neuen Zeichensatz anstelle des alten verwenden soll. Weitere Probleme entstehen daraus, daß der ROM normalerweise nicht für BASIC sichtbar ist. Deshalb muß der ROM eingeblendet werden: Um dies sicher auszuführen, müssen wir den Interrupt herauspoken, damit er nicht verrückt spielt, wenn er herausfindet, daß jemand in den Speicher guckt. Die Vorgehensweise dabei ist wie folgt:

1) Stoppen Sie den Interrupt.
2) Blenden Sie den Zeichen-ROM-Speicher ein.
3) Kopieren Sie die Zeichendefinitionen.
4) Blenden Sie den ROM aus.
5) Starten Sie den Interrupt erneut.

Verwenden Sie POKE 56334, PEEK(56334)AND 251, um den Interrupt anzuhalten, und POKE 56334, PEEK(56334)OR 4, um ihn erneut zu starten.
Verwenden Sie POKE 1, PEEK(1)AND 254, POKE 1, PEEK(1)OR 254, um den ROM-Speicher ein- und auszublenden.
Der Zeichen-ROM beginnt bei 53248 (er liegt unter dem VIC-II-Chip). Wenn Sie die ersten 512 Bytes kopieren, dann werden alle alphanumerischen und Sonderzeichen kopiert, ohne den gesamten Zeichensatz zu kopieren. Bei BASIC kann das eine ganze Weile dauern. Deshalb ist eine COPY-Routine in dieser Situation sehr hilfreich.
Um die Adresse der Zeichendefinition wieder zurückzulegen, verwenden Sie folgendes:

POKE 12288, PEEK (12288)AND 240+X

Dabei ist X eine ganze Zahl zwischen 0 und 15, die auf den Anfang der Zeichenmatrix zeigt – X•1024. Das kann bedeuten, daß Sie BASIC ein wenig verschieben müssen, um Platz zu machen. Weitere Einzelheiten der UDGs können Sie aus dem Programmierhandbuch entnehmen.

# EIN UDG-EDITOR

Das hier aufgelistete Programm erleichtert Ihnen den Entwurf Ihrer eigenen UDGs. Wenn Sie es laufen lassen (RUN), dann gibt es eine kurze Verzögerung, da die alphanumerischen Zeichen kopiert werden und die Zeichenmatrix bewegt wird. Dann wird der Bildschirm gelöscht. Er zeigt Ihnen daraufhin ein 8x8-Quadrat, in dem Sie Ihre eigenen Graphiken erzeugen können, und ein kleines Menü, das die Belegung der Funktionstasten erläutert. Auf dem Quadrat befindet sich ein Cursor, der mit den Cursor-Steuertasten bewegt werden kann. Um ein Quadrat zu füllen, drücken Sie einfach die Leertaste, zur Löschung eines Quadrates verwenden Sie die Taste „←". Die Funktionstasten unterstützen Sie bei Ihrem Entwurf in der folgenden Weise:

F1: Sie erarbeitet die Matrix für die Figur auf dem Entwurfquadrat und druckt die Zahl gegenüber der entsprechenden Zeile.
F2: Diese Option fordert eine Zeicheneingabe an. Tippen Sie einfach ein Zeichen (z.B. „A") ein und drücken Sie RETURN. Die Zeichenmatrix wird dann abgefragt und die Matrix auf das Entwurfsquadrat kopiert.
F5: Sie entspricht F2, außer daß sie das geforderte Symbol überlagert.
F7: Sie wertet das Symbol im Entwurfsquadrat aus und weist es dem POKE-Code zu, der angefordert wird.

Unter Verwendung dieser Funktionen können Sie sich die Zeichen für ein „A" oder „Q" ansehen, und erkennen, wie Commodore sie entworfen hat, Sie können zwei graphische Zeichen mischen, sie selbst unter Verwendung des Cursors verändern und dann Ihren eigenen, persönlichen Zeichensatz erzeugen. Die Verwendung von UDGs in Ihren Spielen hat einen starken Einfluß auf das Endergebnis. Man kann sogar einen UDG-Abschnitt an das Ende eines beliebigen BASIC-Programms anhängen, ohne daß das Programm es bemerkt und ihm so etwas „Pfiff" verleihen. Es gibt keine Entschuldigung dafür, daß Sie Zeichen von jemand anderem kopieren – wenn man alles zusammennimmt, gibt es 2*2*2*2......*2 (64 Zweien) verschiedene Symbole auf einem 8x8-Quadrat. Das ergibt ungefähr $1{,}8*10^{19}$!

**Editor**

```
10 REM UDG EDITOR
15 PRINT"♥BITTE WARTEN..."
20 POKE56334,PEEK(56334)AND254:POKE1,PEE
K(1)AND251
30 FORA=0TO2047:POKE12288+A,PEEK(53248+A
):NEXT
```

```
40 POKE1,PEEK(1)OR4:POKE56334,PEEK(56334
)OR1
50 POKE53272,(PEEK(53272)AND240)+12
60 POKE53281,0:PRINT"♥":POKE53281,246
70 FORA=1TO8:PRINT" ........":NEXT
80 PRINT"■■F1 - BERECHNUNG DER WERTE DER
 MATRIX"
90 PRINT"F3 - ZEICHEN DARSTELLEN"
100 PRINT"F5 - ANDERES ZEICHEN UEBERLAGE
RN"
105 PRINT"F7 - ZEICHEN IN DEN ZEICHENSAT
Z                  EINFUEGEN"
110 P=1065:U=46:POKEP,81
120 GETA$:IFA$=""THEN120
130 V=0:IFA$="■"THENV=1
140 IFA$="■"THENV=-1
150 IFA$="□"THENV=-40
160 IFA$="■"THENV=40
170 IFPEEK(P+V)=46ORPEEK(P+V)=160ORV=0TH
EN190
180 V=0
190 IFV<>0THENPOKEP,U:P=P+V:U=PEEK(P):PO
KEP,81
200 IFA$=" "THENPOKEP,160:U=160
205 IFA$="←"THENPOKEP,46:U=46
210 IFA$="■"THENGOSUB2000
220 IFA$="■"THENGOSUB2000
230 IFA$="■"THENGOSUB3000
240 IFA$="■"THENGOSUB4000
999 GOTO120
1000 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                                    "
1010 RETURN
2000 REM UEBERLAGERUNG
2010 GOSUB1000
2020 INPUT"ZEICHEN";C$
2030 N=PEEK(1913)
2040 FORA=0TO7:D=PEEK(12288+N*8+A)
2050 FORJ=0TO7:S=1065+A*40+J:IFA$="■"THE
NPOKES,46
2060 IF2↑(7-J)=<DTHEND=D-2↑(7-J):POKES,1
```

```
60
2070 NEXTJ,A:U=PEEK(P):RETURN
3000 REM ZEICHENSATZ AENDERN
3010 GOSUB1000:INPUT"NUMMER DES ZEICHENS
";C:IFC<0ORC>255THEN3010
3020 FORA=0TO7:T=0:FORJ=0TO7
3025 IFPEEK(1065+A*40+J)=160THENT=T+2↑(7
-J)
3030 NEXT:POKE12288+C*8+A,T:NEXT
3040 POKE1063,C:POKE55335,14:RETURN
4000 REM WERTEBERECHNUNG
4010 PRINT"█"
4020 FORA=0TO7:T=0
4030 FORJ=0TO7:IFPEEK(1065+A*40+J)=160TH
ENT=T+2↑(7-J)
4040 NEXT:PRINT"███████████████          ██████
";T
4050 NEXT:RETURN
```

## SPRITES

Dieser kurze Abschnitt zielt auf die vorzügliche Sprite-Graphikfunktion des Commodore 64. Um diese bestmöglich zu nutzen, muß zuvor eine Menge Arbeit in den Entwurf der Zeichen investiert werden. Sprites ähneln UDGs insofern sehr, daß jeder Punkt steuerbar ist. Mit 24 Punkten in der Breite (drei Bytes) und 21 Punkten in der Höhe sind sie aber viel größer. Ihre Definitionen werden als die Zahlenkette gespeichert, die sie darstellen, wenn jeder Satz aus 8 Bit als Binärzahl (von links nach rechts) gelesen wird. Es gibt 8 Sprites, und das Handbuch beschreibt ihre Verwendung und ihren Mißbrauch in allen Einzelheiten. Ich liefere Ihnen aber hier einen flexiblen Sprite-Editor.

Das Programm erzeugt eine große Quadrat- und Cursoranordnung, auf der ein Sprite erzeugt werden kann. Der Cursor wird mit den Cursorsteuertasten bewegt. Auf dem großen Quadrat kann die Wiederholungsfunktion mit gutem Erfolg eingesetzt werden. Jede der 8 Funktionstasten dient einem anderen Zweck. Außerdem werden Leertaste und die „←"-Taste verwendet, die Quadrate zeichnen bzw. löschen.

F1: Berechnet die entsprechenden Matrixzahlen und druckt sie in die jeweilige Bildschirmposition.

F2: Läßt das Programm erneut laufen. Dabei wird das Quadrat für einen neuen Entwurf gelöscht.

F3: Sie spiegelt die linke in die rechte Hälfte und ist eine große Hilfe zur Erzeugung von symmetrischen Sprites.

F4: Führt das Programm aus.

F5: „PEN DOWN" – der Cursor hinterläßt eine dauerhafte Spur.

F6: DUMP? YES – die Spritematrix wird berechnet und in die Data-Anweisungen eingegeben – drücken Sie einfach RETURN
NO – das Spritesymbol wird auf Band gespeichert.

F7: „PEN UP" – Der Dauerspureffekt von F5 wird beendet.

F8: Die Daten werden vom Band geladen.

**Sprite-Editor**

```
10 REM SPRITE EDITOR
11 REM
12 REM **PAUL ROPER
15 DIMS(64)
20 PRINT"▚██████████████████████████SPRIT
E-EDITOR"
30 FORA=1TO21:PRINT"....................
....":NEXT
40 P=1024+160+10
50 UC=46
60 GETK$
61 IFK$="█"THENRUN
62 IFK$="█"THEN1000
63 IFK$="█"THEN3000
64 IFK$="█"THEN4000
65 IFK$="█"THEN5000
70 IFK$="█"THENIFPEEK(P+1)<>32THENPOKEP,
UC:P=P+1:UC=PEEK(P)
72 IFK$="█"THENIFPEEK(P-1)<>32THENPOKEP,
UC:P=P-1:UC=PEEK(P)
73 IFK$="█"THEN2000
74 IFK$="□"THENIFPEEK(P-40)<>32THENPOKEP
,UC:P=P-40:UC=PEEK(P)
75 IFK$="█"THENPD=0
76 IFK$="█"THENIFPEEK(P+40)<>32THENPOKEP
```

```
,UC:P=P+40:UC=PEEK(P)
77 IFK$="■"THENPD=1
78 IFK$=" "THENUC=160
79 IFK$="←"THENUC=46
81 IFPDTHENUC=160
90 IFUC=46THENPOKEP,81
100 IFUC=160THENPOKEP,102
110 GOTO60
1000 REM BERECHNUNG UND AUSGABE DER MATR
IX-NUMMERN
1005 PRINT"■"
1010 FORQ=1064TO1864STEP40
1015 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■■■";
1020 FORA=QTOQ+16STEP8
1030 V=128:T=0
1040 FORB=ATOA+7:IFPEEK(B)=160ORPEEK(B)=
102THENT=T+V
1050 V=V/2:NEXT
1060 PRINTT;:NEXT:PRINT
1070 NEXT
1080 GOTO60
2000 REM SPIEGELN DER LINKEN AUF DIE REC
HTE HAELFTE DES SPRITES
2010 FORQ=1064TO1864STEP40
2020 FORA=0TO11
2030 K=Q+A:IFPEEK(K)=102ORPEEK(K)=160THE
NPOKEQ+23-A,160
2040 NEXTA,Q
2050 GOTO60
3000 REM SPRITE AUF KASSETTE SPEICHERN
3005 N=1
3006 POKEP,UC
3008 X=1
3010 FORQ=1064TO1864STEP40:FORA=0TO16STE
P8
3020 V=128:T=0:FORB=0TO7
3030 IFPEEK(Q+A+B)=160THENT=T+V
3040 V=V/2:NEXT:S(X)=T:X=X+1:NEXTA,Q
3045 INPUT"■AUSGABE DER WERTE (J/N)";D$:
IFLEFT$(D$,1)="J"THEN9000
3050 INPUT"■NAME DES SPRITES";N$
```

```
3060 OPEN1,1,1,N$
3070 FORA=1TO63:PRINT#1,S(A):NEXT:CLOSE1
:RUN
4000 REM SPRITE LADEN
4005 N=1
4010 INPUT"♥NAME DES SPRITES";N$
4020 OPEN1,1,0,N$
4030 PRINT"♥■■■■■■■■■■■■■■■■■■■■■■■■■SPR
ITE-EDITOR"
4040 FORA=1TO21:PRINT"..................
......":NEXT
4050 FORQ=1064TO1864STEP40:FORA=0TO16STE
P8
4060 INPUT#1,D:V=128
4070 FORW=0TO7:IFD>=VTHEND=D-V:POKEQ+A+W
,160
4080 V=V/2:NEXT:NEXT:CLOSE1:GOTO60
5000 PRINT"♥PROGRAMM BEENDET.":END
9000 INPUT"STARTZEILE";LS:PRINT"♥";
9010 FORA=1TO63STEP9
9020 L$="":L$=STR$(LS)+"DATA":LS=LS+10
9030 FORB=ATOA+8:L$=L$+STR$(S(B)):IFB<A+
8THENL$=L$+","
9040 NEXT:PRINTL$:NEXT
9050 END
```

# ANHANG B

## MINIASSEMBLER

Egal, ob Sie bereits einen Assembler besitzen oder nicht, Sie werden meinen „Miniassembler" als eine brauchbare Zugabe zu Ihrer Sammlung von Hilfsroutinen empfinden. Er ist in BASIC geschrieben, ist aber sehr kompakt, und enthält ein paar nützliche Eigenschaften einschl. eines schnellen Disassemblers und einer genialen Editiermethode. Der erste Teil dieses Anhangs beschreibt das Programm und seine Anwendung. Außerdem gibt es da noch ein paar Programm-„Werkzeuge", die an den Interrupt angehängt sind und mit den Funktionstasten aufgerufen werden. Der Anhang wird mit einem Maschinencode-Ladeprogramm abgeschlossen, das einen kleinen Werkzeugsatz lädt, der alle vorausgegangenen Eigenschaften beim Interrupt ausführt.
Der Miniassembler verwendet oft DATA-Anweisungen: die Speicherung der Liste mnemonischer Symbole, ihrer Befehlssymbole und des Assemblierprogramms erfolgt unter Anwendung dieser Anweisungen. Das bedeutet ein einfaches Editieren – Sie verwenden wie üblich den flexiblen Bildschirmeditor. Sie können eine Anweisung jeder Zeile haben, oder auch mehrere Anweisungen mit Kommata getrennt. Ihr Programm wird im Bereich der Zeilennummern 20 bis 999 geschrieben. Das gibt Ihnen reichlich Platz. Wenn Sie den Anfang von BASIC nach hinten POKEn wollen, dann sollten Sie dies tun, bevor Sie das Programm laufen lassen. Normalerweise arbeiten BASICAssembler langsam und mühselig. Aber unter Verwendung eines solchen Editors wird das ganze Assemblieren auf einmal ausgeführt und das Programm nimmt von Zeit zu Zeit bei den üblicheren Befehlen Abkürzungen durch die mnemonische Tabelle.

## PROGRAMMBESCHREIBUNG

Nach Eingeben des RUN-Befehls wird die Titelroutine in Zeile 3000 ausgeführt. Es geht dann geradewegs weiter, bis es an dem mnemonischen Ladeprogramm in Zeile 9000 angekommen ist. Wenn die Arrays dimensioniert und die Daten eingelesen sind, dann kehrt die Steuerung nach Zeile 3000 zurück, wo dem Benutzer die Entscheidung Assemblieren oder Diassemblieren abverlangt wird. Das Disassemblieren findet in den Zeilen 2000 bis 2999 statt. Das Assemblieren in den Zeilen 1000 bis 1999. Zuerst wird der Datenzeiger zurückgesetzt (RESTORE) und danach werden nacheinander alle mnemonischen Symbole durchgelesen. STRING-auftrennende Routinen ziehen Einzelheiten, wie Modus, mnemonisches Dreibuchstabensymbol und Operanden heraus. Wird der mnemonische Code mehrfach verwendet, dann werden Abkürzungen eingeleitet. Eventuell auftre-

tende Probleme werden als Fehlermeldung auf dem Bildschirm ausgegeben, und das Assemblieren ist zu Ende, wenn die END-Marke des Programms erscheint. Um die Effizienz des Programms zu erhöhen, werden die Adressiermodi durch ein einzelnes, angehängtes Zeichen direkt hinter dem mnemonischen 3-Buchstaben-Symbol dargestellt. Einzelheiten darüber können Sie aus Tabelle B.1 entnehmen. Es gibt keine Vorkehrungen für die Verwendung von Marken, aber relative Verzweigungen werden dennoch für Sie berechnet. Achten Sie darauf, die DATA-Anweisungen richtig einzutippen – das kostet etwas Zeit.

**Tabelle B.1: PREFIX-Tabelle für Mini-Assembler**

| ADRESSIERMODUS | KENNUNG |
|---|---|
| Unmittelbar | # |
| Absolut | £ |
| Absolut X | X |
| Absolut Y | Y |
| Relativ | ↑ |
| Akkumulator | A |
| Implizit | nur mnemonisches 3-Buchstaben-Symbol |
| Seite-Null X | 1 |
| Seite-Null Y | 2 |
| Seite-Null | Z |
| Indirekt | $ |
| Vorindiziert | ! |
| Nachindiziert | ? |

Der Miniassembler-Befehl ist wie folgt strukturiert:

[Mnemonisches 3-Buchstaben-Symbol]+[Kennung]+[Operand]

Bei impliziten Anweisungen ist nur das 3-Buchstaben-Symbol erforderlich.

## ANWENDUNG

Um den Disassembler anzuwenden, lassen Sie das Programm laufen und antworten „JA". Das Programm fragt jetzt nach einer Startadresse, um die Disassemblierung zu beginnen. Wird diese eingegeben, dann erzeugt die Leertaste jedesmal, wenn sie betätigt wird, eine Anweisung. Sie können auch die Leertaste gedrückt halten. Um das Programm zu verlassen, drücken Sie die RUN/STOP-Taste. GOTO 2000 bringt Sie zum Disassembler zurück.

Der Assembler wird angewendet, indem zuerst der Code in die DATA-Anweisungen mit den Zeilennummern <1000 eingetippt wird und man dann das Programm laufen läßt und den Assembler aufruft. Das Laden und Sichern (LOAD und SAVE) ist einfach: sichern (SAVE) Sie alles zusammen mit Ihrem Code. Es empfiehlt sich, eine dauerhafte Kopie, gefüllt mit leeren DATA-Anweisungen, aufzubewahren. Dieser Befehl wird mit dem mnemonischen Standard-3-Buchstaben-Symbol, einer Moduskennung und dann einem Operand dargestellt. Sehen Sie sich die folgenden Befehle an:

| | |
|---|---|
| LDA#100 | unmittelbares Laden des Akkumulators mit 100 |
| STAX1024 | Speichern des Akkumulators bei 1024+X |
| BCC↑851 | bei nicht gesetztem Übertrag nach 850 springen |
| LDA!251 | vorindiziertes Laden des Akkumulators bei 251 |
| RTS | Rücksprung von der Subroutine. |

Durch den Befehl RUN wird der Code standardmäßig von Speicherstelle 49152 an assembliert. Das kann man durch eine Anweisung ändern, die einfach die Adresse des nächsten zu assemblierenden Befehls neu ausrichtet. Die Anweisung wird wie folgt verwendet:

*820

Dadurch beginnt das Assemblieren in Speicherstelle 820. Der *-Befehl kann mehrfach verwendet werden, so daß Sie mehr als eine Routine gleichzeitig assemblieren können. Das Assemblier-Listing zeigt die Wirkungsweise dieser Befehle durch eine revers dargestellte Feldmitteilung.

Der Miniassembler wirkt ein wenig fremd, aber nach einer kurzen Zeit wird er Ihnen einfach vorkommen. Die folgenden Routinen werden alle im „Mini"-Format gedruckt. Jede Routine führt nützliche Aufgaben im Bereich der Funktionstasten aus. Es sind durch zahlreiche Veröffentlichungen bekannte Eigenschaften des Commodore 64, aber niemand scheint sie jemals für ihren ursprünglich bestimmten Zweck zu verwenden. Ich hoffe, daß diese Routinen Ihnen zeigen, wie Sie sie für Ihre Zwecke nutzbar machen können.

```
10 REM ES FOLGT ASSEMBLER CODE
15 DATA*820
20 DATA LDX#0,STAX1024,INX,BNE↑822,RTS
30 DATA ASLA
1000 REM ASSEMBLER
1010 GOTO3000
1020 RESTORE:P=49152
1025 PRINT"ASSEMBLIERUNG GESTARTET."
1030 READC$:IFC$="*↑*"THENPRINT"ASSEMBLI
ERUNG BEENDET.":END
1035 PRINT:PRINTP,C$"    ";:S=0
1036 IFLEFT$(C$,1)="*"THENP=VAL(RIGHT$(C
$,LEN(C$)-1)):PRINT"[REVERS EIN]P=";P;:G
OTO1030
1040 M$=MID$(C$+CHR$(0),4,1)
1045 I$=LEFT$(C$,4):B$=LEFT$(C$,3)
1065 IFM$=CHR$(0)THEN1081
1070 A$=RIGHT$(C$,LEN(C$)-4)
1071 IFM$<>"↑"THEN1080
1072 V=VAL(A$)-P-2:IFV<0THENV=256+V
1073 IFV<0ORV>255THENPRINT"FEHLER:":PRIN
T"[REVERS EIN]"C$" ZU GROSSER SPRUNG":V=
0
1074 L=V:GOTO1090
1080 W=VAL(A$):H=INT(W/256):L=W-H*256
1081 IFB$="CLC"THENG=24:GOTO1110
1082 IFB$="SEC"THENG=56:GOTO1110
1083 IFB$="JSR"THENS=32:GOTO1090
1084 IFB$="RTS"THENG=96:GOTO1110
1085 IFLEFT$(B$,2)="ST"THENS=129:GOTO109
0
1086 IFLEFT$(B$,2)="LD"THENS=160
1090 FORX=STO255:IFM$(X)=I$THENG=X:X=255
:NEXT:GOTO1110
1095 NEXT:PRINT"FEHLER:[REVERS EIN]"I$"
UNBEKANNT":END
1110 POKEP,G:P=P+1:IFNB(G)=0THEN1030
1120 IFNB(G)=1THENPOKEP,L:P=P+1:GOTO1030
1130 POKEP,L:POKEP+1,H:P=P+2:GOTO1030
```

```
2000 REM DISASSEMBLIERUNG
2010 INPUT"STARTADRESSE";P
2020 M=PEEK(P):M$=M$(M):IFM$="0"THENM$="
[REVERS EIN]UNGUELTIGER CODE":PRINTP,M$:
P=P+1:GOTO2080
2030 IFNB(M)=0THENPRINTP,M$:P=P+1:GOTO20
80
2040 IFNB(M)=1THENPRINTP,M$,PEEK(P+1):P=
P+2:GOTO2080
2050 PRINTP,M$,PEEK(P+1)+PEEK(P+2)*256:P
=P+3
2080 IFPEEK(197)<>60THEN2080
2090 GOTO2020
3000 PRINT"♥***MINI-ASSEMBLER ***QQQ"
3010 PRINT"BITTE WARTEN...":GOSUB9000
3020 INPUT"QQDISASSEMBLIEREN (J/N)";Q$
3030 IFLEFT$(Q$,1)<>"J"THEN1020
3035 PRINT
3040 GOTO2000
9000 DATA*↑*,BRK,0,ORA!,1,0,0,0,0,0,0,OR
AZ,1,ASLZ,1,0,0,PHP,0,ORA#,1,ASLA,1
9010 DATA0,0,0,0,ORA£,2,ASL£,2,0,0,BPL↑,
1,ORA?,1,0,0,0,0,0,0,ORA1,1,ASL1,1
9020 DATA0,0,CLC,0,ORAY,2,0,0,0,0,0,0,OR
AX,2,ASLX,2,0,0
9030 DATAJSR£,2,AND!,1,0,0,0,0,BITZ,1,AN
DZ,1,ROLZ,1,0,0,PLP,0,AND#,1,ROLA,0
9040 DATA0,0,BIT£,2,AND£,2,ROL£,2,0,0,BM
I↑,1,AND2,1,0,0,0,0,0,0,AND1,1
9050 DATAROL1,1,0,0,SEC,0,ANDY,2,0,0,0,0
,0,0,ANDX,2,ROLX,2,0,0
9060 DATARTI,0,EOR!,1,0,0,0,0,0,0,EORZ,1
,LSRZ,1,0,0,PHA,0,EOR#,1,LSRA,0,0,0
9070 DATAJMP£,2,EOR£,2,LSR£,2,0,0,BVC↑,1
,EOR?,1,0,0,0,0,0,0,EOR1,1,LSR1,0
9080 DATA0,0,CLI,0,EORY,2,0,0,0,0,0,0,EO
RX,2,LSRX,2,0,0
9090 DATARTS,0,ADC!,1,0,0,0,0,0,0,ADCZ,1
,RORZ,0,0,0,PLA,0,ADC#,1,RORA,0,0,0
9100 DATAJMP$,2,ADC£,2,ROR£,2,0,0,BVS↑,1
,ADC?,1,0,0,0,0,0,0,ADCZ,1,RORZ,1,0,0
```

```
9110 DATASEI,0,ADCY,2,0,0,0,0,0,0,ADCX,2
,RORX,2,0,0
9120 DATA0,0,STA!,1,0,0,0,0,STYZ,1,STAZ,
1,STXZ,1,0,0,DEY,0,0,0,TXA,0,0,0
9130 DATASTY£,2,STA£,2,STX£,2,0,0,BCC↑,1
,STA?,1,0,0,0,0,STY1,1,STA1,1,STX1,1
9140 DATA0,0,TYA,0,STAY,2,TXS,0,0,0,0,0,
STAX,2,0,0,0,0
9150 DATALDY#,1,LDA!,1,LDX#,1,0,0,LDYZ,1
,LDAZ,1,LDX,1,0,0,TAY,0,LDA#,1,TAX,0
9160 DATA0,0,LDY£,2,LDA£,2,LDX£,2,0,0,BC
S↑,1,LDA?,1,0,0,0,0,LDY1,1
9170 DATALDA1,1,LDX2,1,0,0,CLV,0,LDAY,2,
TSX,0,0,0,LDYX,2,LDAX,2,LDXY,2,0,0
9176 DATACPY#,1,CMP!,1,0,0,0,0;CPYZ,1,CM
PZ,1,DECZ,1,0,0,INY,0,CMP#,1,DEX,0,0,0
9177 DATACPY£,2,CMP£,2,DEC£,2,0,0,BNE↑,1
,CMP?,1,0,0,0,0,0,0,CMP1,1,DEC1,1
9178 DATA0,0,CLD,0,CMPY,2,0,0,0,0,0,0,CM
PX,2,DECX,2,0,0
9184 DATACPX#,1,SBC!,1,0,0,0,0,CPXZ,1,SB
CZ,1,INCZ,1,0,0,INX,0,SBC#,1,NOP,0
9185 DATA0,0,CPX£,2,SBC£,2,INC£,2,0,0,BE
Q↑,1,SBC?,1,0,0,0,0,0,0
9190 DATASBC1,1,INC1,1,0,0,SED,0,SBCY,2,
0,0,0,0,0,0,SBCX,2,INCX,2,0,0
9200 RESTORE
9210 READA$:IFA$<>"*↑*"THEN9210
9220 DIMM$(255),NB(255),Q$(10,20)
9230 FORA=0TO255:READM$(A),NB(A):NEXT
9240 RETURN
```

## BENUTZERDEFINIERBARE FUNKTIONSTASTEN

Diese Routine stellt die Funktionstaste F1 so ein, daß sie maximal 10 Zeichen durch einmaliges Drücken eintippt. Dieser „Ausdruck" wird von 820 an gespeichert und mit einer 13 beendet, die einen Wagenrücklauf bewirkt. Mit Hilfe dieser Routine können Sie die Taste so belegen, daß sie den Bildschirm löscht und den Befehl LIST oder RUN für das Programm aufruft. Sie wird an den Interrupt angehängt, so daß Sie sie, wenn Sie einmal installiert ist, vergessen und anschließend das Ladeprogramm auf NEW setzen können.

```
*49152
LDA£197
CMP#4
BNE↑49174
LDX#0
LDAX820
STAX631
INX
CMP#13
BNE↑49161
STXZ198
JMP£59953
```

Angenommen, wir wollen die Taste so einstellen, daß sie „LIST" gefolgt von einem RETURN ausgibt. Verwenden Sie die folgende Routine:

```
T$="LIST"+CHR$(13)
FOR A=1TO LEN(T$):POKE819+A,ASC(MID$(T$,A,1)):
NEXT A
```

Vergessen Sie nicht, den Interruptvektor wieder neu auszurichten:

```
POKE788,0:POKE789,192
```

## DYNAMISCHER HALT

Diese Routine ermöglicht Ihnen das Anhalten der gesamten Maschine, egal, was diese gerade tut (außer bei Bandbetrieb). Wird F3 gedrückt, dann begibt sich die Maschine in einen „dynamischen Halt", so lange, bis die Leertaste gedrückt wird. Das ist besonders nützlich bei der Überprüfung oder Fehlerbeseitigung eines Programm-Listings, bei der die CTRL-Taste nicht betätigt wird. Die interne Uhr stoppt jedoch während des Halts.

```
LDAZ197
CMP#5
BNE↑49167
JSR£65439        Tastatur abfragen
LDAZ197
CMP#60           Leertaste gedrückt?
BNE↑49158
JMP£59953
```

Wie bei allen diesen Routinen muß der Anschluß des Interrupts wieder hergestellt werden.

## QUOTE-MODUS-ABSCHALTUNG

Kennen Sie das Gefühl? Sie editieren eine Zeile, die eine PRINT-Anweisung enthält, und irgendwie (durch Eingabe einer ungeraden Zahl von Anführungszeichen) gelangen Sie in den Quote-Modus (=Anführungszeichen-Modus) und alle Cursorsteuerungen scheinen gegen Sie zu arbeiten. Diese Routine bringt durch Betätigen von F5 die Welt wieder in Ordnung und bringt Sie aus dem Quote-Modus heraus:

```
LDAZ197
CMP#6
BNE↑49162
LDA#0
STAZ212          Verläßt den Quote-Modus
JMP£59953
```

## GERÄUSCH- UND SPRITEABSCHALTUNG

Beim Herumprobieren mit Geräuschen und Sprites läßt Sie Ihr Gerät oftmals sitzen, mit einem Bildschirm voller Sprites und einem kopfwehträchtig wimmernden Fernseher. Eine schnelle Antwort ist die Kombination RUN/STOP+RESTORE. Diese löscht jedoch alle nützlichen Informationen vom Bildschirm und stellt den Interrupt zurück (wodurch diese Routinen abgeschaltet werden). Die folgende Routine belegt die F7-Taste mit der Geräusch- und Spriteabschaltung. Nur ein Tastendruck und schon kehrt Ruhe ein.

```
         LDAZ197
         CMP#3
       ,-BNE↑49166
      /  LDA#0
     |   STA£54296      Schaltet die Geräusche ab
      \  STA£53269      Schaltet die Sprites ab
       `>JMP£59953
```

## KLEINER WERKZEUGSATZ

Alle diese Routinen beginnen bei 49152. Um sie zu mischen, müssen Sie die Verzweigungen neu ausrichten und sie miteinander verketten. Es ist eine gute Übung zur Überprüfung Ihrer Fähigkeiten. Das folgende Ladeprogramm liest einfach ein Maschinencode-Programm ein, das alle 4 Routinen miteinander kombiniert und den Zeiger wieder auf den Interrupt ausrichtet. Ist dieses erst einmal gemacht, so können Sie den NEW-Befehl für das Programm eingeben und mit dem Programmieren normal weitermachen – nur daß Sie jetzt einen kleinen Werkzeugsatz an Bord haben.

```
1     REM**KLEINER WERKZEUGSATZ**
2     REM ENABLE:POKE 788,0:POKE 789,192
3     REM DISABLE:POKE 788,49:POKE 789,234
10    P=49152
20    READ D:POKE P,D:P=P+1:GOTO20
6000  DATA 165,197,201,4,240,15,201,5,240,29,201,6,240,37,201,3,240
6010  DATA 43,76,49,234,162,0,189,52,3,157,119,2,232,201,13,208,245
6020  DATA 134,198,76,49,234,32,159,255,165,197,201,60,208,247,76,49
6030  DATA 234,169,0,133,212,32,228,255,76,49,234,169,0,141,24,212
6040  DATA 141,21,208,76,49,234
```

Geben Sie "RUN" ein.
Warten Sie auf "?OUT OF DATA ERROR".
Freigabe wie oben.
Geben Sie den Befehl NEW für das Programm ein.

F1: Tippe Zeichenfolge in 820.
F3: Gebe dynamischen Halt ein.
    Fortfahren mit Leertaste.
F5: Verlassen des Zitiermodus.
F7: Geräusch- und Spriteabschaltung.

Der Werkzeugsatz kann beliebig angeordnet werden – alle Verzweigungen sind relativ. Nebenbei bemerkt, wenn Sie einen Joystick an Port 1 angeschlossen haben, wird der „Feuer"-Knopf zu einer weiteren Leertaste.

# ANHANG C

# FORTSCHRITTLICHE MASCHINENKENNWERTE

## DIE TASTATUR UND IHR PUFFER

Der Abschnitt über den Dialog (Kapitel 9) enthält einen kurzen Exkurs über die Tastatur und deren Bedienung. Ein tieferes Verständnis ihrer Betriebsweise ermöglicht jedoch die Verwendung bestimmter Tricks bei der Programmierung. In einigen Routinen dieses Buches verwende ich den Puffer, um bestimmte Dinge zu erreichen. Einfacher ausgedrückt, der Puffer ist nichts weiter als eine Liste von Tastaturbetätigungen, die noch nicht auf dem Bildschirm ausgegeben sind.
Jede 1/60 Sekunde läuft unser Freund, der Interrupt, herum und sieht sich die Tastatur an. Wurde eine neue Taste gedrückt, dann wird dieses im Puffer aufgezeichnet und der Pufferzeiger erhöht. Blinkt der Maschinencursor, dann leert eine andere Routine den Puffer. Daher füllt sich der Puffer auf, solange der Cursor nicht blinkt – bis er maximal 10 Zeichen enthält. Wenn Sie ein Programm auflisten und auf der Tastatur herumklimpern, dann erhalten Sie nach Beendigung der Auflistung eine Liste der von Ihnen getroffenen Tasten.
Der Puffer läuft von 631 bis 640. Der Pufferzeiger ist 198. Da wir wissen, wo er sich befindet, hält uns nichts davon ab, den Puffer mit Zeichen zu füllen. Die Zeichen werden im ASCII-Format gespeichert und das ist genau die Technik, die für die Definition der Funktionstasten verwendet wird. Ein Befehl wie LIST wird zusammen mit einem RETURN (Code 13) in den Puffer eingegeben. Wenn Sie glauben, daß der Tastaturpuffer Probleme verursacht, dann kann seine Größe reduziert werden durch POKEn einer neuen Größe in die Speicherstelle 649. Überschreiten Sie auf keinen Fall eine Größe von 10, da der Puffer sich sonst in gefährliches Gebiet (die Betriebssystemvariablen) begibt.
Die folgenden Routine POKEt M$ in den Puffer.

```
FOR A=1 TO LEN(M$) : POKE 630+A,ASC(MID$(M$,A,1)) :
NEXT
POKE 198,LEN(M$)
```

Eine clevere Anwendung des Puffers ermöglicht es den Programmen, sich selbst zu ändern, indem sie über eine END- oder STOP-Anweisung hinüberkommen. Sie führen dann „automatische Rücksprünge" über eine neue Zeile und einen CONT- oder GOTO-Befehl aus, der die Ausführung wieder aufnimmt. Dieses Thema schweift aber ziemlich weit von dem eigentlichen Zweck dieses Buches ab.

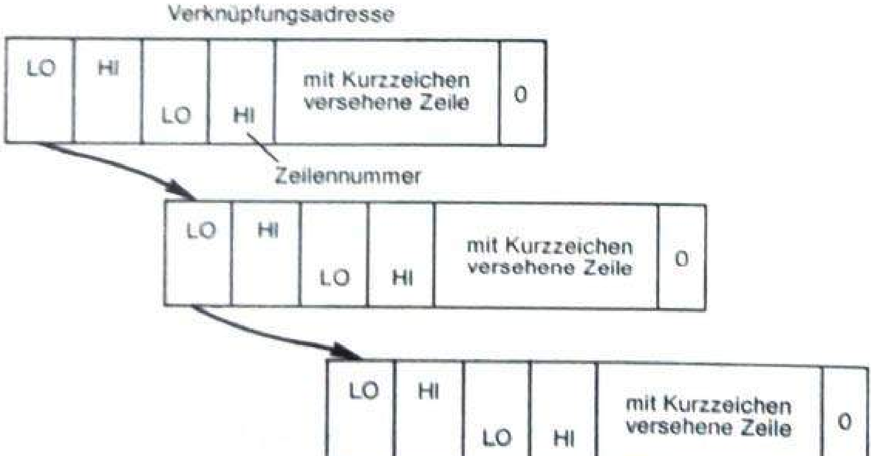
# TEXT- UND VARIABLENSPEICHERUNG IN BASIC

Obwohl dieser Abschnitt streng genommen nichts mit dem Maschinencode zu tun hat, besitzt er dennoch eine direkte Verbindung zu ihm. Außerdem ist er nicht im Programmierhandbuch enthalten. Von besonderer Bedeutung ist die Beschreibung der Gleitkommavariablen, da dies das Format ist, in dem Daten über die USR(X)-Funktion weitergegeben werden. Die Art und Weise, in der Programme und Variablen im C 64 gespeichert werden, sollte es Ihnen erlauben, Ihre eigenen Routinen zu schreiben, z.B. eine Umnumerierungs- und Suchroutine.

Das Programm beginnt in Speicherstelle 2049. Jede Zeile wird als ein Datensatz in einer Datei gespeichert und in steigender Reihenfolge der Zeilennummern abgelegt. Diese Datensätze haben folgendes Format.

Die ersten beiden Bytes enthalten das, was man eine „Verknüpfungsadresse" nennt. Diese beiden Bytes bilden die Startadresse des nächsten Datensatzes. Dadurch kann man schnell durch ein Programm jagen, ohne jedes einzelne Zeichen suchen zu müssen. Die nächsten beiden Bytes enthalten die Zeilennummer. Das ist der Grund dafür, das die größe der Zeilennummern begrenzt ist. Der daran anschließende Datensatz ist einfach eine Zeichenfolge aus Bytes, die mit einem Null-Byte abgeschlossen wird. Das nächste Byte ist das niedrigwertige Byte der nächsten Verknüpfungsadresse. Schlüsselworte, wie GOTO, GOSUB und PRINT werden durch „Kurzzeichen" dargestellt. Dieses sind Einzelbytewerte, die das Schlüsselwort eindeutig angeben und dadurch viel Speicherplatz sparen. In Abbildung C.1 ist diese Struktur erklärt.

**Abbildung C.1: Textspeicherung in BASIC**

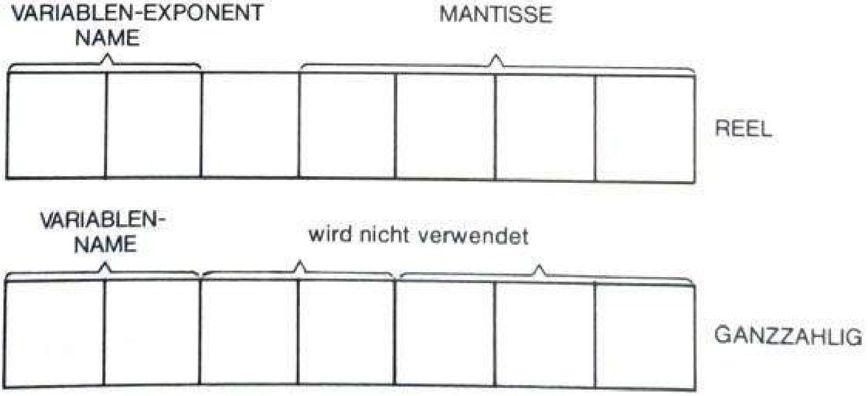


Die Speicherstellen 45 und 46 enthalten den Anfang der BASIC-Variablen. Es gibt davon viele Arten: reelle Zahl, ganze Zahl, reelles Array, ganzzahliges Array, String, String-Array. Für jede Variablenart gibt es eine spezielle Darstellung. Reellwertige Variablen werden in 7 Bytes gespeichert. Die ersten beiden Bytes enthalten die ASCII-Werte des Namens. Deshalb verbraucht eine Variable genau soviel Speicherplatz wie zwei Zeichen zusammengenommen. Die nächsten 5 Bytes bilden den Wert der Variablen. Es wird die wissenschaftliche Schreibweise verwendet. Sie erlaubt, daß jede Zahl in der folgenden Weise dargestellt werden kann:

**$\pm a * 10^{x}$, dabei ist $0 < a < 1$ und x eine ganze Zahl.**

Der einzige Unterschied besteht darin, daß beim C 64 nicht die 10 verwendet wird. Da er ein Computer ist, verwendet er die 2. Das „x" oder der „Exponent" wird im ersten Byte gespeichert. Um die Verwendung negativer Zahlen zu ermöglichen, wird der Exponent so gespeichert, als wären 129 zu seinem wirklichen Wert hinzugezählt. Deshalb beträgt der tatsächliche Bereich für x −129 bis 126. Machen Sie nicht den Gedankenfehler, daß ein negativer Exponent eine negative Zahl bedeutet – er bedeutet einfach, daß die Zahl sehr klein ist ($10^{-3} = 0{,}001$).
Die „Mantisse" (der „a"-Teil) wird in den nächsten 4 Bytes gespeichert. Sie wird in der binärcodierten Dezimalform gespeichert. Das bedeutet, daß jedes Byte in ein Paar aus 4 Bitstückchen aufgeteilt wird, von dem jedes eine Zahl darstellt. Dadurch wird eine 8-Bit-Genauigkeit erreicht. Das erste Bit des ersten Bytes der Mantisse enthält das Vorzeichen. Der Vollständigkeit halber beschreibe ich auch die ganzzahlige Variable. Für sie wird ebenfalls eine 7-Bit-Darstellung verwendet. Die ersten beiden enthalten den Namen, der sich von dem der „reellen Variable"

**Abbildung C.2: Realer und ganzzahliger Variablen**



dadurch unterscheidet, daß er ASCII-Codes plus 128 beträgt. Die nächsten beiden Bytes enthalten die niederwertigen und höherwertigen Werte, gefolgt von 3 Scheinnullen. (Sie sparen nur bei ganzzahligen Arrays). Abbildung C 2 stellt die Methode der Variablenspeicherung graphisch dar.

## DIE CIAs

Anfangs gab es den VIA – den vielseitigen Schnittstellenadapter (Versatile Interface Adaptor). Es war (und ist immer noch) als ein bemerkenswert flexibler peripherer Chip für Eingabe/Ausgabe und Zeitsteuerung.

Der Commodore 64 enthält nicht nur einen, sondern zwei CIAs – komplexe Schnittstellenadapter (Complex Interface Adaptor). Es sind extrem leistungsfähige Chips, die aber viel zu selten benutzt werden. Einer der CIAs ist für die IRQ-Interrupts zuständig. Jeder CIA enthält eine komplette Uhr mit einem 12-Stundensystem, 2 anschließbare 16 Bit-Timer, 1 Alarmeinrichtung und eine vollständige Handshaking-Fähigkeit. Für die Zwecke dieses Buches erläutere ich nur die Timer und ihre Steuerung für die IRQ-Planung. Das Programmierhandbuch beschreibt sie nur kurz, aber um ihnen voll gerecht zu werden, sind die Datenblätter der Hersteller erforderlich.

Die beiden Timer sind als A und B bekannt. Für jeden Timer gibt es einen 16-Bit-Signalspeicher, von dem aus sie abwärts zählen. Erreicht der Timer 0, dann wird der Wert erneut in den Signalspeicher geladen und ist dadurch bereit zum nächsten Count-down. Erreicht ein Timer 0, so wird außerdem ein IRQ erzeugt. Jeder Timer kann im Einmal- oder Dauerbetrieb arbeiten. Im Einmalbetrieb stoppt der Timer, wenn er 0 erreicht, und lädt den Signalspeicher neu. Er wartet dann auf ein Startsignal. Im Dauerbetrieb wird der Wert immer wieder neu geladen. Die Timer zählen jeden Zyklus der Uhr der Hauptzentraleinheit CPU (1000000 Hz). Beim Einschalten wird Timer A initialisiert, um jede 1/60 Sekunde einen Interrupt zu erzeugen.

Beide Timer werden von einem Steuerregister gesteuert. Die 5 niederwertigen Bits führen die folgenden Funktionen durch:

Bit 0: START/STOP. **Dieses Bit startet und stoppt den Timer. Ist es gesetzt, dann läuft der Timer.**

Bit 1: **Hier nicht relevant.**

Bit 3: Gesetzt = Ein**malbetrieb, nicht gesetzt = Dauerbetrieb.**

Bit 4: Wird dieses Bit **gesetzt, dann lädt es sofort den Startwert in den** Signalspeicher, **egal, was der Timer gerade macht.**

Damit Sie die Timer und nicht die Interrupts zur Zeitsteuerung verwenden können, wird eine Interruptmaske bereitgestellt. Sie verhindert einfach, das ein IRQ von der Quelle abgegeben wird. Die Timer können einzeln maskiert werden. Beim Einschalten erzeugt der Timer A Interrupts, während Timer B maskiert ist. Das Maskenregister ist 56333. Bit 7 bestimmt, ob die Maske gelöscht oder gesetzt wird. Ist Bit 7 gelöscht, dann beseitigt irgendeine 1, die in 56333 geschrieben wird, die Fähigkeit der entsprechenden Quelle, einen Interrupt zu erzeugen. Das Umgekehrte gilt für eine 0. Timer A ist Bit 0 und B ist Bit 1. Timer B kann dazu verwendet werden, nicht die Taktimpulse, sondern die Bereichsunterschreitungen von Timer A zu zählen. Das bedeutet, falls Timer A eine Interruptmaske eingeschaltet hat, können die beiden Timer zusammenarbeiten, um so einen 32 Bit-Timer zu erzeugen. Das Zählverfahren wird durch den Zustand der Bits 5 und 6 in 56335 festgelegt: 00 bedeutet „Zähle Taktimpulse", 10 bedeutet „Zähle die Bereichsunterschreitungen von Timer A".
Um einen Signalspeicherwert zu setzen, POKEn Sie ihn einfach in die niederwertige/höherwertige Timerspeicherstelle. Das PEEKen dieser Stellen liefert den gegenwärtigen Zählerstand. Die folgende Tabelle gibt alle Speicherstellen an, die wir für CIA 1 benötigen:

| | |
|---|---|
| 56324 | niederwertiges Byte des Timers A |
| 56325 | höherwertiges Byte des Timers A |
| 56326 | niederwertiges Byte des Timers B |
| 56327 | höherwertiges Byte des Timers B |
| 56333 | IRQ-Steuerregister |
| 56334 | Steuerregister Timer A |
| 56335 | Steuerregister Timer B |

Wir wollen jetzt die Tätigkeit der Timer beobachten. Der blinkende Cursor blinkt mit Hilfe des Interrupts. Probieren Sie den folgenden POKE-Befehl aus und beobachten Sie den Cursor:

```
POKE 56325,20
```

Probieren Sie die Wiederholungsfunktion mit den Cursortasten aus – bemerken Sie etwas? Der Cursor blinkt dreimal so schnell! Wollen Sie Ihre Interruptroutinen schneller arbeiten lassen, dann können Sie das auf diese Weise erreichen. Vergessen Sie aber nicht, daß TI$ sich ebenfalls dreimal so schnell ändert. Wir wollen jetzt Timer B einbeziehen:

```
POKE 56335,1          Einschalten
POKE 56327,64         Signalspeicher setzen
POKE 56333,127+1+2    IRQ-Maske entfernen.
```

Das Ergebnis davon ist, daß Timer A alle 1/180 Sekunde unterbricht, während B nur 1/3-mal so schnell arbeitet. Probieren Sie herum, es ist die einzige Art, etwas zu lernen. Versuchen Sie nicht, alle IRQ-Leitungen abzuschalten, dies würde u. a. den Verlust der Dialogfähigkeit der Tastatur bewirken.
Der zweite CIA besitzt alle diese Eigenschaften, ist aber nicht an die IRQ-Leitung angeschlossen. Beide CIAs können diese Impulszüge mit großer Leichtigkeit durch die externen Anschlußstellen (Ports) senden und empfangen. Ich habe bis jetzt die Verwendung eines CIAs zur Erzeugung eines komplexen Impulszuges nur anklingen lassen. Ihrer Verwendung könnte ein ganzes Buch gewidmet werden. Sie sollten jedoch nicht hier aufhören. Wenn der IRQ der Schlüssel zum C 64 ist, dann ist CIA 1 der Schlüssel zum IRQ.

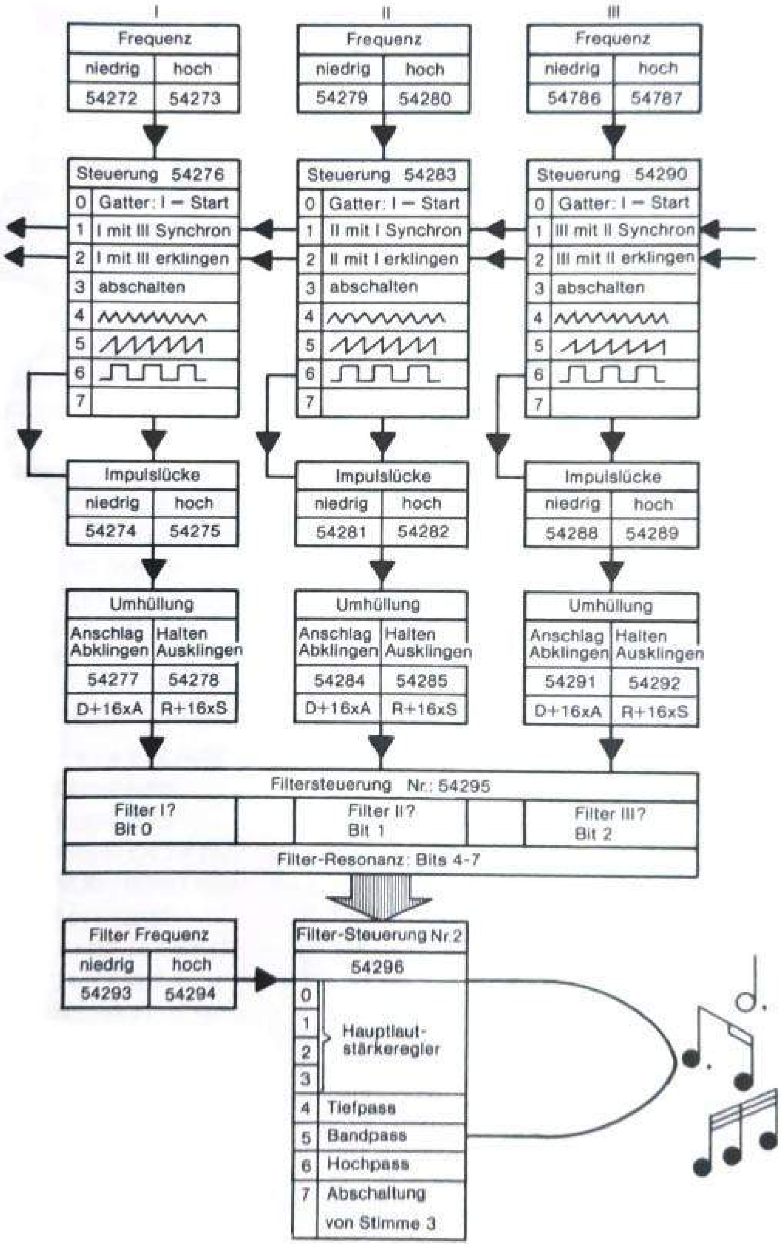
## SID- UND VIC-SPEICHERBELEGUNGSPLÄNE

Obwohl das Programmierhandbuch informativ und kurz gefaßt ist, muß man viel in ihm blättern, wenn man Programme schreibt. Das hängt mit der Komplexität der SID- und VIC-II-Chips zusammen. Es dauert lange, bis man alle Speicherplätze auswendig kennt. Weiterhin kann das Format nicht schnell durchgeprüft werden. Ich gebe Ihnen hier einige meiner Tabellen wieder. Sie zeigen übersichtlich, was welche Speicherstelle tut und wie man sie verwendet. Das SID-Programm (Abbildung C.3) zeigt insbesonders die komplexe Anordnung von Stimmen und Filtern, deren Struktur sonst schwer verständlich ist.
Anmerkung: Diese Tabellen setzen voraus, daß Sie den Betrieb der SID- und VIC-II-Chips bereits verstehen. Sie versuchen nicht, ihre Funktion und Verwendung zu erklären.

**Tabelle C.1: Sprite-Referenztabelle**

| Sprite | Steuerwert | X | Y | Farbe | M.Zeiger |
|---|---|---|---|---|---|
| 0 | 1 | 53248 | 53249 | 53287 | 2040 |
| 1 | 2 | 53250 | 53251 | 53288 | 2041 |
| 2 | 4 | 53252 | 53253 | 53289 | 2042 |
| 3 | 8 | 53254 | 53255 | 53290 | 2043 |
| 4 | 16 | 53256 | 53257 | 53291 | 2044 |
| 5 | 32 | 53258 | 53259 | 53292 | 2045 |
| 6 | 64 | 53259 | 53260 | 53293 | 2046 |
| 7 | 128 | 53260 | 53262 | 53294 | 2047 |

## Abbildung C.3: SID-Referenztabelle



Verwenden Sie den Steuerwert, um die Bits in den folgenden Spritesteuerregistern zu setzen:

| | |
|---|---|
| 53264 | höchstwertiges Bit der X-Koordinate des Sprites |
| 53271 | senkrechte Ausdehnung des Sprites |
| 53277 | waagerechte Ausdehnung des Sprites |
| 53278 | Protokoll der Kollisionen zwischen Sprites |
| 53279 | Protokoll der Kollisionen zwischen Sprites und Hintergrund |
| 53280 | Farbe des Randes |
| 53281 | Farbe des Bildschirm-Hintergrundes. |

| | | | |
|---|---|---|---|
| 0 | Schwarz | 1 | Weiß |
| 2 | Rot | 3 | Blaugrün |
| 4 | Violett | 5 | Grün |
| 6 | Blau | 7 | Gelb |
| 8 | Orange | 9 | Braun |
| 10 | Hellrot | 11 | Grau Nr. 1 |
| 12 | Grau Nr. 2 | 13 | Hellgrün |
| 14 | Hellblau | 15 | Grau Nr. 3 |

# ANHANG D

## LASERBIKE

Dies ist ein Spiel, in dem ein paar Ideen aus diesem Buch verwendet wurden. Sie müssen mit einem Fahrrad in einer zylindrischen Welt herumfahren, von der immer nur ein kleiner Teil sichtbar ist. Beim Fahren hinterlassen Sie eine nicht überschaubare, brennende Spur – ihre Überquerung bedeutet den Tod.
Das Ziel des Spiels ist das Sammeln aller Schlüssel ('$'-Zeichen), die dem Schwierigkeitsgrad entsprechend gefordert werden und das Verlassen durch das Zeit-Tor ('#'-Zeichen). Um sie wachsam zu halten, sind Bomben auf dem Spielfeld verteilt. Explodiert eine, dann ist es aus. Der Navascan leitet sie, so daß Sie die Bomben entschärfen können. Sie haben 3 Fahrräder, wenn Sie aufhören, diese zu bewegen, dann können sie überhitzt werden. Zusatzpunkte können erlangt werden, indem man auf dem Weg Früchte ('*') aufsammelt. Die folgenden Tasten bewegen das Fahrrad: ',' nach links, '.' nach rechts, 'Q' nach oben, 'A' nach unten. So wie es aussieht, ist das Spiel interessant, aber nicht unbedingt brilliant: Es sollte Ihnen ja auch nur einfach zeigen, welche Dinge mit Unterstützung des Maschinencodes möglich sind. Es werden 2 Maschinencoderoutinen verwendet: die Fensterroutine, die das Fenster auf dem Bildschirm projeziert, und eine Auffüllroutine, die den Spielbereich nach jedem Spiel löscht. Vielleicht hätten Sie das Spiel gerne etwas interessanter, aber „Laserbike" wurde an nur einem Nachmittag geschrieben, um die Fensterroutine auszuprobieren. Verbesserungen könnten z.B. durch das Anhängen eines UDG-Schalters an den Interrupt sein, damit die Spur wirklich brennt. Außerdem könnten Sie den graphischen Aufbau verbessern, denn es wurde nur die Standardgraphik des Commodore verwendet.

```
0 REM *****LASER BIKE****
10 GOSUB9000
12 GOSUB9500
15 DEFFNR(X)=INT(X*RND(1)+1)
30 GOSUB1000:REM SPIEL
31 PRINT""
35 M$="SPIELENDE":GOSUB10000
40 PRINT"PUNKTE:"SC
50 IFSC>HSTHENPRINT"[REVERS EIN]
EIN NEUER REKORD !!!":HS=SC
55 FORT=1TO3000:NEXT
60 GOTO30
1000 REM SPIEL
1010 LV=0:SC=0:LB=3:TP=0:CT=1653
1011 S$="
     "
```

```
1012 S1$=S$+"██":B$=S$+"[REVERS EIN]----
--":B1$=S1$+"[REVERS EIN]------"
1013 L$=S$+"[REVERS EIN]LINKS-":R$=S$+"[
REVERS EIN]RECHTS"
1014 U$=S1$+"[REVERS EIN]-OBEN-":D$=S1$+
"[REVERS EIN]UNTEN-"
1015 CC=0:LV=LV+1:GOSUB6000:GOSUB8000
1020 GOSUB5000:REM STATUS
1025 NK=3+INT(LV/5)
1030 FORA=1TOLV*10:POKE49266+FNR(59)+FNR
(63)*60,42:NEXT
1032 FORA=1TOLV*50:POKE49266+FNR(59)+FNR
(63)*60,46:NEXT
1033 BL=49266+FNR(59)+FNR(63)*60:IFPEEK(
BL)<>32THEN1033
1034 POKEBL,30:BD=255
1040 FORA=1TO(NK*2+1):L=49266+FNR(59)+FN
R(63)*60:IFPEEK(L)<>32THENA=A-1:NEXT
1050 POKEL,36:NEXT:POKEL,35
1060 REM TASTATURABFRAGE
1070 PL=52000:D=1
1080 K=PEEK(197):C(0)=C(D):D=0:IFK=47THE
ND=2
1085 POKEPL+305,160
1090 IFK=44THEND=1
1100 IFK=62THEND=3
1110 IFK=10THEND=4
1115 IFD=0THENTP=TP+.2:IFINT(TP-.05)<INT
(TP)THENCT=CT+1:GOSUB5000
1120 PL=PL+V(D):H=INT(PL/256):L=PL-H*256
1121 Q=PEEK(PL+305):IFD=0THENQ=1
1122 IFQ=32THEN1130
1123 IFQ=42THENSC=SC+15*LV:GOSUB5000:POK
E54276,0:POKE54276,33
1124 IFQ=36THENCC=CC+1:SC=SC+100:GOSUB50
00:POKE54276,0:POKE54276,17
1125 IFQ=35ANDNK=CCTHENGOSUB9600:GOTO101
5
1126 IFQ=35THENRETURN
1127 IFQ=30THENGOSUB5900
1128 IFQ=46THENGOSUB5500
```

```
1129 IFQ=160THENGOSUB5500
1130 POKE253,L:POKE254,H
1135 POKEPL+305,C(D)
1140 SYS49152
1145 D1=(PL+305)-BL:V1=INT(D1/60):H1=D1-
V1*60:IFH1<0THENH1=H1+60
1146 IFH1=0THENPRINTB$:GOTO1149
1147 IFH1<30THENPRINTL$:GOTO1149
1148 PRINTR$
1149 IFV1=0THENPRINTB1$:GOTO1160
1150 IFV1<0THENPRINTD$:GOTO1160
1151 PRINTU$
1160 BD=BD-1:IFBD=0THENGOSUB5500
1999 GOTO1080
5000 PRINT"███████████";LV
5005 POKECT,81:IFCT=1656THENTCT=1653:TP=
0:FORA=CTTOCT+3:POKEA,173:NEXT:GOTO5500
5010 PRINT"█████████";NK
5020 PRINT"██████████";CC
5030 FORA=1157TO(1155+2*LB)STEP2:POKEA,1
07:NEXT:POKEA,32
5035 PRINT"▣"TAB(10)HS:PRINT"▣"TAB(31)SC

5040 RETURN
5500 REM ZERSTOERUNG DES 'BIKE'
5505 POKE54276,0:POKE54276,129
5510 FORT=1TO3000:NEXT
5520 LB=LB-1:TP=0:PL=PL-V(D):POKEPL+305,
32:D=0
5530 IFLBTHENGOTO5000
5540 GOTO31
5900 BL=49266+FNR(59)+FNR(63)*60:IFPEEK(
BL)<>32THEN5900
5910 POKEBL,30:BD=255:RETURN
6000 REM BILDSCHIRMAUFBAU
6010 PRINT"┘HI-SCORE:                  PUN
KTE:    "
6020 PRINT"[REVERS EIN]/\/\/\/\/\/\/\/\/\/\/\/\/
\/\/\/\/\/\/\/\/\/\/\/\/\/\"
6025 PRINT"
          "
```

```
6026 PRINT"
          "
6030 PRINT"[WEISS]LEVEL:0      [REVERS EIN
]XXXXXXXXXXXXXX"
6040 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6050 PRINT"NO.KEYS:    [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6060 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6070 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6080 PRINT"NO.HELD:    [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6090 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6100 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6110 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6120 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6130 PRINT"                  [REVERS EIN]XX[REV
ERS AUS]          [REVERS EIN]XX[REVERS
AUS]"
6140 PRINT"                  [REVERS EIN]XXXXXX
[REVERS AUS]▼[REVERS EIN]XXXXXX[REVERS
AUS]"
6150 PRINT"                     [REVERS EIN]NAV
ASCAN"
6160 PRINT"                      [REVERS EIN]
    [REVERS AUS]"
```

```
6170 PRINT"                        [REVERS EIN]
    [REVERS AUS]"
6180 PRINT"       >   >   >   [REVERS EIN]
    [REVERS AUS]  <   <   <"
6190 PRINT"                        [REVERS EIN]
    [REVERS AUS]"
6200 PRINT"                        [REVERS EIN]
    [REVERS AUS]"
6210 PRINT"■[GELB]  [REVERS EIN]***LASER
-BIKE***  (C)1983 PAUL ROPER"
6220 PRINT"■■■■■■■■■■■■■■■"
6230 PRINTTAB(25)"    [REVERS EIN]◤  ◥"
6240 PRINTTAB(25)"  [REVERS EIN]◤┌TEMP┐◥
"
6250 PRINTTAB(25)" [REVERS EIN]LO+----+H
I
6260 PRINTTAB(25)"   [REVERS EIN]└─────┘"

6999 RETURN
8000 REM INITIALISIERUNG
8010 SYS820
8020 FORA=49205TO49264:POKEA,160:NEXT
8040 FORA=53105TO53164:POKEA,160:NEXT
8050 RETURN
8500 FORA=1TOLV*10:POKE49266+FNR(59)+FNR
(63)*60,42:NEXT
8510 FORA=1TO4:L=49266+FNR(59)+FNR(63)*6
0
8520 IFPEEK(L)<>32THENA=A-1:NEXT
8530 POKEL,36:NEXT
8540 POKEL,35:RETURN
9000 REM MASCHINENCODE FUER DAS 'FENSTER
'
9010 DATA169,253,133,251
9020 DATA169,4,133,252
9030 DATA162,0,160,0
9040 DATA177,253,145,251
9050 DATA200,192,10,208
9060 DATA247,232,24,165
9070 DATA253,105,60,133
9080 DATA253,165,254,105
```

```
9090 DATA0,133,254,24
9100 DATA165,251,105,40
9110 DATA133,251,165,252
9120 DATA105,0,133,252
9130 DATA224,10,208,214,96,*
9140 RESTORE:P=49152
9150 READD$:IFD$="*"THEN9200
9160 POKEP,VAL(D$):P=P+1:GOTO9150
9200 DATA107,1,115,-1,113,-60,114,60
9210 FORA=1TO4:READC(A),V(A):NEXT
9220 P=820
9230 READD:IFD=-1THENRETURN
9240 POKEP,D:P=P+1:GOTO9230
9250 DATA169,53,133,253
9260 DATA169,192,133,254
9270 DATA160,0,169,32
9280 DATA145,253,230,253
9290 DATA208,2,230,254
9300 DATA165,253,201,172
9310 DATA208,240,165,254
9320 DATA201,207,208,234,96,-1
9500 REM TONERZEUGUNG
9510 FORS=54272TO54295:POKES,0:NEXT:POKE
54296,15
9520 POKE54277,12:POKE54278,0
9530 POKE54273,10:RETURN
9600 REM GERAEUSCH FUER DEN ZEIT-TUNNEL
9610 FORA=30TO100STEP25:POKE54276,0:POKE
54276,129
9620 FORB=ATOA+35:POKE54273,B:POKE53280,
FNR(16)-1:NEXT:NEXT
9630 POKE53280,254:RETURN
10000 PRINTTAB(13);
10010 FORA=1TOLEN(M$):PRINTMID$(M$,A,1);
:FORT=1TO150:NEXT:NEXT
10020 PRINT"[illegible]":RETURN
11000 RETURN
```

# INDEX DER ROUTINEN


| Routine | Beschreibung | Stelle |
|---|---|---|
| **Alternatives Spritesystem** | Spritesystem auf Zeichengrundlage | Kapitel 9, Alternatives Spritesystem |
| **Arrayaddition** | Addiert einen Vektor zu einem Array/einer Tabelle | Kapitel 10, Flottenbewegungen |
| **Arraylöschung** | Löscht ein Element in einem Array | Kapitel 10, Flottenbewegungen |
| **Attributblinker** | Läßt automatisch beliebige Bildschirmzeilen blinken | Kapitel 8, Invertierung und Explosionen |
| **Arraysuche** | Sucht ein Array bzw. eine Tabelle | Kapitel 10, Flottenbewegungen |
| **Blockauffüllung** | Einfache Füllroutine | Kapitel 9, Speicherauffüllung |
| **Bomben-aktualisierung** | Tastet den Bildschirm ab und bewegt Bomben um ein Zeichen nach unten | Kapitel 9, Bomben-aktualisierung |
| **CHRGET-Keil** | Ein besonderer Zugang zum Maschinencode | Anhang A, Eintritt in den Maschinencode |
| **Wiederanlauf nach Absturz** | Systemrückstellung ohne Programmverlust | Anhang A, Wiederanlauf nach Absturz |
| **Verzögerung** | 3 verschiedene Verzögerungsroutinen | Kapitel 9, Verzögerungen |
| **Dynamischer Halt** | Blockiert und startet den Computer auf Anforderung | Anhang B, Dynamischer Halt |

| | | |
|---|---|---|
| **Defender-Landschaftsfenster** | Projiziert Landschaften | Kapitel 10, Spiele mit großem Umfang |
| **Funktionstastenzuweisung** | Weist F1 eine Textzeichenfolge zu | Anhang B, Funktionstasten |
| **Interruptmelodien** | Spielt Melodien unter Anwendung des Interrupt | Kapitel 10, Geräusche |
| **Invertierung** | 2 Routinen, um den Bildschirm zum Blinken zu bringen | Kapitel 9, Invertierung und Explosionen |
| **Zeilenlöschen** | Rollhilfsroutine | Kapitel 9, Rollen |
| **Speicherkopie** | Kopiert den Speicher aus einem Bereich in einen anderen | Kapitel 9, Kopieren des Speichers |
| **Miniassembler** | | Anhang B, Miniassembler |
| **Kleiner Werkzeugsatz** | 4 handliche Funktionstastenzuweisungen | Anhang B, Miniassembler |
| **Rechteckige Auffüllung** | Füllt ein Rechteck mit einem anderen Kode als den Hauptbereich | Kapitel 9, Speicherauffüllung |
| **Quote-Modus-Abschaltung** | Stellt die Funktion der Cursor-Steuertasten wieder her | Anhang B, Quote-Modus-Abschaltung |
| **Bildschirmabtastung** | Tastet einfach den Bildschirm ab | Kapitel 6 |
| **Rollen** | 3 Rollroutinen | Kapitel 9, Rollen |

| | | |
|---|---|---|
| **Geräusch- und Spriteabschaltung** | Schaltet Geräusche und Sprites ohne Datenverluste ab | Anhang B, Geräusch- und Spriteabschaltung |
| **Spiralförmige Bildschirmauffüllung** | Füllt den Bildschirm von innen nach außen, als ob eine Spirale gezeichnet wird | Kapitel 6 |
| **Spritekollisionen** | Beseitigung von Spritekollisionen | Kapitel 10, Sprites |
| **Spriteansteuerung** | Steuert die Sprites 1–7 auf Sprite 0 zu | Kapitel 10, Nichtlineare Bewegung |

Unter dem Obertitel 'Programmierung in Maschinensprache auf dem Commodore 64' erscheinen in der Commodore Sachbuchreihe mehrere Bücher zu verschiedenen Aspekten der Maschinencode-Programmierung. Sie stammen von verschiedenen Autoren und sind unabhängig voneinander entstanden, so daß jeder Band eine abgeschlossene Einheit darstellt.

In diesem Band zeigt Paul Roper, daß Maschinencode-Programmierung auf dem Commodore 64 leicht zu erlernen ist. Er beginnt mit der Anwendung von Maschinencode in BASIC-Programmen und entwickelt dann die speziellen Techniken für den 6502-Prozessor.

Zum eigentlichen Thema, der Spieleprogrammierung, gelangt der Autor auf drei Wegen: Nach einer Einführung in die Technik des Spiele-Entwurfs werden einige komplexe Maschinencode-Spiele entwickelt und beschrieben. Schließlich werden sehr nützliche Subroutinen angeboten, mit deren Hilfe der Leser seine eigenen Spiele garnieren kann.

Bildschirmrollen und Fenstertechnik oder Sprite-Bewegungen sind nur Beispiele der reichhaltigen Ideen, die ausführlich beschrieben werden. Dazu werden sehr brauchbare Maschinencode-Subroutinen entworfen, eingeschlossen ein Assembler und ein Editor.

**Commodore**

Commodore GmbH
Lyoner Straße 38
D-6000 Frankfurt/M. 71

Commodore AG
Aeschenvorstadt 57
CH-4010 Basel

Commodore GmbH
Kinskygasse 40–44
A-1232 Wien

Nachdruck, auch auszugsweise, nur mit schriftlicher Genehmigung von Commodore.
Artikel-Nr. 556427/4.85 Änderungen vorbehalten ISBN 3-89133-014-6